IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 16 November 2023, accepted 23 December 2023, date of publication 25 December 2023,
date of current version 4 January 2024.

Digital Object Identifier 10.1109/ACCESS.2023.3347422

RESEARCH ARTICLE

D(HE)at: A Practical Denial-of-Service Attack on the Finite Field Diffie–Hellman Key Exchange

SZILÁRD PFEIFFER[1] AND NORBERT TIHANYI[2,3], (Member, IEEE)
[1]BalaSys IT Ltd., 1117 Budapest, Hungary
[2]Department of Computer Algebra, Eötvös Loránd University (ELTE), 1117 Budapest, Hungary
[3]Technology Innovation Institute (TII), Abu Dhabi, United Arab Emirates

Corresponding author: Norbert Tihanyi (ntihanyi@inf.elte.hu)

The work of Szilárd Pfeiffer was supported by Balasys IT Ltd. The work of Norbert Tihanyi was supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund Financed under the TKP2021-NVA Funding Scheme under Project TKP2021-NVA-29.

ABSTRACT In this paper, D(HE)at, a practical denial-of-service (DoS) attack targeting the finite field Diffie-Hellman (DH) key exchange protocol, is presented, allowing remote users to send non-public keys to the victim, triggering expensive server-side DH modular-exponentiation calculations. The attack was disclosed in November 2021 with an assigned CVE-2002-20001 number. Additionally, the ''long exponent'' issue, an implementation flaw in cryptographic libraries where unreasonably large private keys are used, deviating from the recommended NIST guidelines, and making D(HE)at more effective, is presented. This issue was disclosed in November 2022 with an assigned CVE-2022-40735 number. A thorough analysis of the D(HE)at attack, along with proof of concept code that has the potential to compromise all existing protocols employing DH key exchange, such as TLS or SSH, is presented in this paper, highlighting the necessity of additional security measures for effective safeguarding. The potential of reaching full 100% CPU utilization by the D(HE)at attack is demonstrated, even on the most up-to-date operating systems, without any significant computation on the client side. With minimal bandwidth and a low request rate per second (rps), the D(HE)at attack can be carried out against target machines from a single laptop. In this study, the consequences of these issues are explored, and a comparative security and performance analysis is conducted among the most commonly used general-purpose cryptographic libraries, including OpenSSL, BoringSSL, LibreSSL, GnuTLS, NSS, Mbed TLS, OpenJDK, Oracle JDK, and WolfSSL. Based on Shodan measurements, it has been found that 87% of servers worldwide support DH key exchange in the SSH protocol, and according to our scan, 55% of the top 1 million websites support DH in TLS. As a result of this study, it is recommended that developers and administrators consider exclusively enabling Elliptic Curve Diffie-Hellman (ECDH), a significantly more efficient protocol, in their server configurations.

INDEX TERMS Diffie–Hellman key exchange, D(HE)at attack, key exchange protocol, DoS.

I. INTRODUCTION

Key Exchange protocols are a fundamental aspect of modern cryptography and play a crucial role in ensuring the security of communication over networks. The primary goal of key exchange protocols is to allow two or more parties to establish a shared secret value over an untrusted channel, which can then be used to encrypt and decrypt messages exchanged between them. The first publicly available such protocol was proposed by Merkle [33] in 1974. Merkle's puzzle protocol allows two parties to agree on a shared secret value using only symmetric cryptography. The main problem with Merkle's puzzle protocol is that it only provides a quadratic gap between the running time of the honest parties and the adversary. In 2009, Barak and Mahmoody [3] showed that the quadratic gap is the best possible if we treat symmetric ciphers as a black box oracle, i.e., that every protocol in the random oracle model where the participants make n oracle queries can be broken with high probability by an adversary making $O(n^2)$ queries. The quadratic gap and the large

The associate editor coordinating the review of this manuscript and approving it for publication was Sedat Akleylek.

amount of data transferred between parties make Merkle's protocol ineffective.

The Diffie–Hellman (DH) key exchange protocol, introduced in 1976 by Whitfield Diffie and Martin Hellman [10], is one of the first practical key exchange methods. In the DH key exchange, a client and a server agree on common group parameters, namely a prime number p and a generator $g \in \mathbb{Z}_p^*$. Here, g generates a cyclic subgroup $G \leq \mathbb{Z}_p^*$ with a prime order q. Then both parties select a private key $a \in \mathbb{Z}_q$ and $b \in \mathbb{Z}_q$. The client calculates its public key and sends $A \equiv g^a \pmod{p}$ to the server. The server, in turn, calculates its public key $B \equiv g^b \pmod{p}$ and sends it back to the client. Both parties compute the shared key $k \equiv (g^a)^b = (g^b)^a = g^{ab} \pmod{p}$. The Computational Diffie–Hellman (CDH) assumption states that computing the shared key g^{ab} from $g^a, g^b \pmod{p}$ is computationally difficult. Throughout the paper, we call (p, g) as the group parameters, (A, B) are the public keys, and (a, b) are the private keys. In the paper, we use the terms private key and exponent interchangeably for (a, b). Although DH key exchange can also be performed using elliptic curve groups, our focus is solely on the "mod p" case, also known as finite field Diffie–Hellman (FFDH).

A. DIFFIE–HELLMAN EPHEMERAL (DHE) AND FORWARD SECRECY

In the standard DH key exchange, the same private keys are reused for multiple sessions. This means that if an adversary were to somehow obtain the private key, they would be able to decrypt all past and future communications, meaning that it cannot provide forward secrecy. However, by using ephemeral keys, a new key pair is generated for each session, and the private key is discarded after the shared key calculation. In this paper, when we mention "DH key exchange", we are specifically referring to the finite field Diffie-Hellman Ephemeral (DHE) key exchange mechanism.

DH key exchange is widely used for establishing session keys in various protocols such as TLS 1.0 [8], SSH 2.0 [83], IKEv1 [20], IKEv2 [24] or QUIC [77]. TLS 1.3 was defined in RFC 8446 [69] and released to the public in August 2018. Due to the importance of forward secrecy offered by ephemeral keys, RSA key exchange is no longer available in TLS 1.3, leaving only DHE and Elliptic Curve Diffie–Hellman Ephemeral (ECDHE) protocols as the available choices.

B. GLOBAL SUPPORT OF DH KEY EXCHANGE

Based on our most recent analysis of the top 1 million websites [32] as of May 2023, we found that 55% of these sites utilize the DH key exchange protocol during the TLS negotiation (see Figure 1). The widespread usage of the SSH protocol on the internet can be assessed in a similar manner. According to the data gathered from Shodan [73], we found that 87% of servers worldwide offer support for DHE within the SSH protocol.

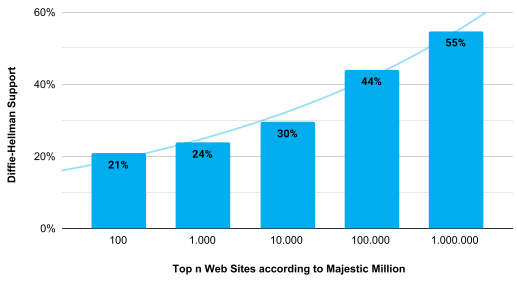

FIGURE 1. Diffie–Hellman support in the Top 1 million domains.

C. RECOVERING THE SHARED KEY

Despite its widespread use, if not implemented correctly, the DH key exchange is vulnerable to various known attacks that can compromise confidentiality or integrity. Man-in-the-middle attacks or the small subgroup confinement attack are just a few examples that are often used against discrete logarithm-based key agreement protocols [31], [80]. For instance, in 2015 Adrian et al. published the Logjam [1] attack against the DH key exchange protocol. The vulnerability allows an attacker to downgrade the encryption strength and potentially decrypt secure communications. The main goal of these attacks is to recover the shared secret value, which can subsequently be used to gain unauthorized access to the encrypted data (confidentiality) or modify data in an unauthorized manner (integrity). However, the third aspect of the CIA triad (availability), is often disregarded and underestimated.

D. RESPONSIBLE DISCLOSURE - D(HE)at AND LONG EXPONENT ATTACK

In November 2021, the D(HE)at attack [37] was officially disclosed as a theoretical threat that has been assigned an official CVE-2002-20001[1] number. We have communicated with numerous vendors, alerting them about the presence and the significance of this attack impacting modern operating systems. While measuring the performance of various implementations, we discovered the "long exponent" issue [38] occurring in multiple cryptographic libraries, to which the identifier CVE-2022-40735 has been allocated. In response to these threats, many vendors and software companies implemented or announced mitigation strategies. To mention a few of the most notable ones; In November 2022, the OpenSSL Team applied a new patch mitigating CVE-2022-40735 from version 3.0.6. In January 2023, Oracle reduced the private key sizes in Java implementations from version 17.0.6. The Cybersecurity & Infrastructure Security Agency (CISA) of the United States issued an Alert Code ICSA-22-314-10 [2], indicating that Siemens products (SCALANCE W1750D) are impacted. F5 Product Development has issued the K83120834 [13] advisory solution. Synology addressed CVE-2002-20001 in their Mail Server from version

[1]CVE-2002-20001 note: The reason why the 2002 CVE assignment was given, even though it was published in 2021, will be discussed in Section III.

1.7.4-10659 [23]. Ubuntu announced CVE-2022-40735 on its webpage [38] and various distributions are currently undergoing evaluation.

1) CONTRIBUTIONS

The main contributions of this paper can be summarized as follows:

- The D(HE)at attack is presented emphasizing its extensive applicability and its consequent impact. It is confirmed through our findings that every protocol employing DH key exchange is susceptible, including well-known ones such as TLS 1.2, TLS 1.3, SSH 2.0, IKEv1 IKEv2, and QUIC. This issue is demonstrated in practice, revealing that this vulnerability extends to all primary operating systems supporting DH key exchange, even those with all current security patches applied.
- The ''long exponent'' issue, a significant implementation flaw in cryptographic libraries, which relates to the employment of large private keys in the Diffie–Hellman key exchange, is disclosed.
- The consequences of the aforementioned attacks are evaluated, and a comprehensive performance and security analysis of their impact on the most widely used cryptographic libraries including OpenSSL, BoringSSL, LibreSSL, GnuTLS, NSS, Mbed TLS, OpenJDK, WolfSSL, is provided. Additionally, a framework is released that can be used to measure the speed of DH key generation across various libraries, aiding administrators during the mitigation phase.
- It is demonstrated that the D(HE)at attack can be carried out even from a single laptop with particularly low bandwidth and request count to achieve 100% CPU utilization on the server side. This makes D(HE)at particularly effective in request count compared to other methods [27], [86].

The paper is organized as follows: Section II will introduce the basic notations and provide an overview of some well-known attacks against the DH key exchange protocol. Section III discusses, the anatomy of the D(HE)at attack and the ''long exponent'' issue. Section IV investigates the impact of these attacks, providing a detailed performance and security comparison of well-known libraries. Section V summarizes the different attack scenarios, while in Section VI we highlight possible strategies to mitigate these attacks. We conclude that it is not possible to completely mitigate the D(HE)at attack at the protocol level, due to the fundamental nature of the modular exponential calculations, however, we are determined to present practical solutions to minimize the impact.

II. PRELIMINARIES

In this section, we provide an overview of the essential notations, concepts, and background that will be used throughout the paper.

A group $(G, \circ)$ is a set G equipped with a binary operation $\circ$, which satisfies the following conditions: closure $\forall a, b \in G, a \circ b \in G$, associativity $\forall a, b, c \in G, (a \circ b) \circ c = a \circ (b \circ c)$, existence of an identity element $\exists e \in G$ such that $\forall a \in G, e \circ a = a \circ e = a$, and presence of an inverse element $\forall a \in G, \exists a^{-1} \in G$ such that $a \circ a^{-1} = a^{-1} \circ a = e$. An Abelian group is a group having the following additional property: $\forall a, b \in G, a \circ b = b \circ a$. The order of a finite group (G is finite) is its cardinality, i.e., the number of its elements in the group and it is denoted by $|G|$. A group $(G, \circ)$ is said to be cyclic if it has a generator g such that for any $a \in G$, there exists an integer n such that $a = g^n$. If g generates all elements of the group $(G, \circ)$, g is a generator and we say it generates $(G, \circ)$. For the sake of simplicity, throughout this paper, we will use $\mathbb{Z}_p^*$ referring to the group composed of the set $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ with multiplication modulo p.

A. DIFFIE HELLMAN PROBLEM

Let G be a cyclic group with order p, and let $g \in G$ be a generator of this group. Then the Diffie–Hellman problem (DHP) involves finding the value of $z = g^{ab}$ for a given $x = g^a$ and $y = g^b$, where a and b are independently and uniformly chosen from the set $\{0, \cdots, p-1\}$. This problem is a key component of the Diffie–Hellman key exchange algorithm.

B. DISCRETE LOGARITHM PROBLEM

Computing the exponent a from the expression g^a for a given base g, where a has been uniformly and randomly chosen from the group, is called the Discrete Logarithm Problem (DLP). Computing the discrete logarithm is the only known method for solving DHP, however, it has not been proved that no other methods exist. The description of the DH key exchange protocol can be seen in Algorithm 1.

Algorithm 1 Diffie–Hellman Key Exchange Protocol

Group parameters: p prime and g generator.

procedure DH
  CLIENT picks a random $a \in \mathbb{Z}_p^*$
  CLIENT computes $A \equiv g^a \pmod p$
  CLIENT sends A to *SERVER*
  SERVER picks a random $b \in \mathbb{Z}_p^*$
  SERVER computes $B \equiv g^b \pmod p$
  SERVER sends B to *CLIENT*
  Both parties compute $k \equiv (g^a)^b = (g^b)^a \pmod p$
  return k shared secret value
end procedure

C. EXPENSIVE MODULAR EXPONENTIATION

Modular exponentiation forms the foundation for numerous public-key cryptographic algorithms and also play an important role in DH key exchange. Directly computing the modular exponentiation $x^k \pmod m$ can be computationally inefficient, especially for large values of x and

k. howeverer, an optimized method can be implemented based on the observation that $x^2 \pmod m$ can be computed as $[(x \pmod m) \times (x \pmod m)] \pmod m$. This method called modular exponentiation by squaring, which has a complexity of $\mathcal{O}(\log k)$, allows for a faster and more memory-efficient computation of modular exponentiation. Although implementing modular reduction operations to keep the numbers smaller requires additional computational overhead, the reduced size of the numbers makes each operation faster, resulting in significant time and memory savings.

1) PARI/GP FOR CRYPTOGRAPHIC CALCULATIONS

PARI/GP [76] is an open-source computer algebra system. Throughout this paper, we will showcase examples using PARI/GP, which is widely used in cryptographic computations and number theory analyses. The fast modular exponentiation method can be easily implemented in PARI/GP which can be seen in Listing 1.

```
powermod(x,k,m)=lift(Mod(x,m)^k);
```

LISTING 1: Fast modular exponentiation in PARI/GP.

Consider to calculate the following huge number $100^{2^{32}} \pmod{2^{25}+1}$. The efficiency of the fast modular exponentiation method becomes apparent. The traditional method takes 3.21 minutes in one CPU thread of a 2017 MacBook Pro using approximately 8 GB of memory in PARI/GP. In contrast, using the exponentiation by squaring method `powermod(x,k,m)` results in a calculation time of less than 1 ms, without requiring significant memory usage. During this paper when examining the key generation speeds of different libraries that utilize modular exponentiation, we will use PARI/GP 2.15.2 as our reference point, ensuring an easy comparison for the research community.

D. ATTACK SURFACES OF DH KEY EXCHANGE

Before examining the intensive server-side DH modular-exponentiation attack, we highlight some fundamental security issues that could potentially threaten the reliability of the DH key exchange protocol. These examples can serve as a useful starting point for readers to gain a deeper understanding of the original D(HE)at attack.

1) SMALL P MODULUS

The security provided by the DH key exchange protocol is heavily dependent on the size of the modulus p. When the prime number p is too small, it is possible to check all possible solutions and solve the discrete log problem quickly. For instance, consider a 64-bit prime number $p = 16292513333391787979$, with $g = 2$ as a primitive root modulo p. The DH key exchange protocol's PARI/GP implementation is shown in Listing 2.

The private key selected by the client is $a = 2012590787486009163$ and the associated public key is $A \equiv g^a \pmod p = 10675403142324386337$. The protocol results in identical keys (k1 and k2) for the CLIENT and SERVER, indicating a shared secret value. The shared secret value is represented by the group element $g^{ab} = 10101250519266596601$.

```
powermod(x,k,m)=lift(Mod(x,m)^k);
p = 16292513333391787979;
g = lift(znprimroot(p)); # 2
a = random(p);           # 2012590787486009163
A = powermod(g,a,p);     # 10675403142324386337
b = random(p);           # 2376326677702921708
B = powermod(g,b,p);     # 3083091892892334750
k1 = powermod(B,a,p);    # 10101250519266596601
k2 = powermod(A,b,p);    # 10101250519266596601
k1 == k2;                # True
```

LISTING 2: PARI/GP example for DH key exchange.

Solving the discrete logarithm problem and deriving the private key a from the public key $g^a \pmod p$ for such a small 64-bit p modulus can be easily achieved, using the `znlog()` PARI/GP command, as shown in the following code snippet:

```
znlog(A,Mod(g,p)) #2012590787486009163
```

LISTING 3: Discrete logarithm computation.

This operation took only 149 ms for a 2017 MacBook Pro. This example clearly underscores the importance of selecting a sufficiently large modulus p to maintain a robust level of security in the DH key exchange protocol. The size of p directly affects the strength of the encryption, making it critical for maintaining the integrity and confidentiality of the data being protected. Among others, NIST provides specific recommendations [4] (see Table 1) for the minimum size of p in order to achieve various levels of security. According to NIST guidelines, to achieve security levels of 80, 112, 128, 192, and 256 bits, one needs to select a prime with the size of at least 1024, 2048, 3072, 7680, and 15360 bits respectively. Therefore it is crucial to choose a large enough p modulus to ensure appropriate bits of security.

2) PRIVATE EXPONENT SIZE

Here we would like to clarify a frequently misunderstood aspect related to the size of the private keys a, b. Increasing

TABLE 1. Comparable security strengths of a symmetric block cipher and asymmetric-key algorithms.

Security Strength	Symmetric Key Algorithm	FFC (DSA, DH, MQV)	IFC (RSA)	ECC (ECDSA, EdDSA, DH, MQV)
≤ 80	2TDEA	$L = 1024$ $N = 160$	$k = 1024$	$f = 160$-223
112	3TDEA	$L = 2048$ $N = 224$	$k = 2048$	$f = 224$-255
128	AES-128	$L = 3072$ $N = 256$	$k = 3072$	$f = 256$-383
192	AES-192	$L = 7680$ $N = 384$	$k = 7680$	$f = 384$-511
256	AES-256	$L = 15360$ $N = 512$	$k = 15360$	$f = 512+$

the size of the private keys beyond a certain threshold, while keeping the modulus p unchanged, would not enhance the security of the protocol. Table 2 provides a comparison of international guidelines (including NIST SP 800-57) concerning the sizes of private keys associated with various modulus values.

TABLE 2. Recommendations for private exponent sizes for various modulus.

Guideline \ Modulus	2048	3072	4096	6144	8192
	Related private key sizes to modulus				
NIST SP 800-57 Part 1 [4]	224	256	292*	348*	394*
RFC 3526 Estimate 1 [28]	220	260	300	340	380
RFC 3526 Estimate 2 [28]	320	420	480	540	620
RFC 3766 [48]	2 × final key size				
RFC 4419 [14]	2 × key material size				
RFC 4535 [22]	RFC 3526				
RFC 7919 [17]	225	275	325	375	400

* estimated value

In the DH key exchange protocol, the public key is represented as $g^a \pmod p$. The discrete logarithm problem can be solved efficiently in $\mathcal{O}(\sqrt{n})$ time using methods like Giant Step/Baby Step and Pollard's Rho, instead of the naive $\mathcal{O}(n)$ approach. To achieve 128-bit security, one must select a prime modulus of 3072 bits and a corresponding private key of 256 bits according to NIST recommendations. Only expanding the size of the private keys without proportionally increasing the modulus p does not yield stronger security, but causes significant performance degradation. For instance, using the NIST-suggested 256-bit private key along with a 3072-bit modulus can be 10 times faster compared to the case when the public key size is the same as the private key size (long exponent). The performance difference can be as much as 20 times using long and short exponents in the case of an 8192-bit modulus size. In the prior example, if we increase the size of the private key a and b to, for instance, 2048-bit, the difficulty of solving the discrete logarithm problem remains the same, irrespective of this increase in the private key size. This attack scenario (large private key, small modulus) can effectively be illustrated by the following code snippet:

```
a = random(2^2048); A = powermod(g,a,p);
b = random(2^2048); B = powermod(g,b,p);
k1 = powermod(B,a,p);
k1==powermod(B,znlog(A,Mod(g,p)),p)  # True
```

LISTING 4: Discrete log attack against DH key exchange.

In our test environment, even with a 2048-bit private key, the `znlog(A,Mod(g,p))` computation can be executed by an attacker in under 200 ms, highlighting the significance of the issue.

3) LARGE MODULUS AND SMOOTH NUMBERS

The security of the key exchange protocol depends greatly on the choice and size of the group parameters (p, g) as well as the minimum size of the private keys (a, b). In order to ensure secure key exchange, it is essential to choose a large modulus according to the NIST recommendation. howeverer, this alone is not sufficient to guarantee security. In 1978, Stephen C. Pohlig and Martin E. Hellman introduced a method [51] for solving the discrete logarithm problem in a finite cyclic group in $\mathcal{O}(\log^2 p)$ time if $p-1$ has only small prime factors. This method is known as the Pohlig–Hellman algorithm and can be used to recover information about exponents if the order of the subgroup generated by g has small factors. The Pohlig–Hellman algorithm is an extension of the Baby-Step Giant-Step algorithm and is particularly efficient for groups with a small prime order. The algorithm pseudocode can be seen in Algorithm 2.

Algorithm 2 Pohlig–Hellman Algorithm

procedure Pohlig–Hellman(p, g, $h \in G$)
  $n \leftarrow p-1$
  factorize n as $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$
  for i = 1 to k **do**
    $h_i \leftarrow h^{n/p_i^{e_i}} \pmod p$
    $g_i \leftarrow g^{n/p_i^{e_i}} \pmod p$
    $x_i \leftarrow$ solution of $g_i^{x_i} = h_i \pmod p$ using the Baby-Step Giant-Step algorithm
  end for
  $x \leftarrow$ solution of $x \equiv x_i \pmod{n/p_i^{e_i}}$ using the Chinese Remainder Theorem
  return x
end procedure

At this point, we also aim to illustrate the fundamental principles of the algorithm with an example where the modulus is large enough. Let us generate a large prime modulus p, such that $p-1$ only has small prime factors. To obtain the modulus p, the first 171 prime numbers were multiplied and $+1$ was added to the result i.e. $p = (2 \cdot 3 \cdot 5 \cdot \ldots \cdot 1019) + 1$. The following command in PARI/GP, `p=prod(i=1,171,prime(i))+1` was used to produce the 1410-bit prime modulus. The Pohlig–Hellman algorithm can be used efficiently to find the discrete logarithm of the public key A if the largest prime factor of $p-1$ is small, which is 1019 in our example. By solely observing the public key $A \equiv g^a \pmod p$, an attacker can determine the discrete logarithm of A within a time frame of 1.7 seconds by utilizing the already mentioned `znlog(A,Mod(g,p))` PARI/GP command. The fact that the recovered value from the public key A and the randomly chosen private key a are identical indicates that the attack was successful. The experiment can be seen in Listing 5.

```
p=prod(i=1,171,prime(i))+1; #1410-bit prime
isprime(p) # True
g=lift(znprimroot(p)) #2
a = random(p) #Random Secret key
A = powermod(g,a,p); #Public key
a==znlog(A,Mod(g,p)) # Recovered secret key
```

LISTING 5: Calculating Discrete logarithm of a smooth $p-1$.

The same method can be employed to conduct a man-in-the-middle attack when the condition $p-1 = qr$ is

satisfied, where r consists of small prime factors exclusively. In the original group $\mathbb{Z}_p^*$ that we are working with, there is a subgroup of order q where the computation of discrete logarithm in q is difficult, as well as a subgroup of order r where it is relatively easy. The keys can be recovered (mod r) as first described by Oorschot and Wiener [80] in 1996.

The CIA triad is a fundamental concept in information security that consists of three key elements: confidentiality, integrity, and availability. The demonstrated attacks including the small modulus attack and the small subgroup attacks using the Pohlig–Hellman algorithm, are all related to the confidentiality and integrity elements of the CIA triad. The third aspect of the CIA triad (availability), is often disregarded and underestimated.

III. THE D(HE)at ATTACK (CVE-2002-20001)

To prevent attacks such as the small subgroup attack, it is commonly recommended to choose a "safe" prime for the value of p in the DH key exchange. A safe prime is of the form $2q+1$, where q is also a prime number. Additionally, the generator g should be selected in a way that generates a group of order q modulo p. By choosing p and g in this manner, the security of the DH key exchange is strengthened against small subgroup attacks. Modern cryptographic libraries often utilize pre-defined group parameters (p, g) in their implementations of the DH key exchange protocol. This approach helps to mitigate the risk of weak parameter selection by ensuring that the parameters used in the key exchange are well-established and secure.

As an illustration, RFC 7919 [17] employs four distinct safe primes. For example the 2048-bit group is defined as $p = 2^{2048} - 2^{1984} + \lfloor 2^{1918} \cdot e + 560316 \rfloor \cdot 2^{64} - 1$ where the generator is $g = 2$ and the group size is $q = (p-1)/2$ is also a prime number. Verifying the generated subgroup is straightforward, as it can be done by checking if $2^q \mod p \equiv 1$. Applying a large prime modulus can effectively safeguard the confidentiality and integrity of the key exchange protocol, howeverer, it also introduces a new attack surface that can potentially compromise the availability of the involved hardware and software components.

Under normal circumstances, the DH key exchange algorithm requires the same amount of resource consumption from the parties participating in the key agreement process, because each party must perform the same operations to calculate their public keys and the shared secret value. In 2002, Jean-Francois Raymond and Anton Stiglic briefly mentioned [68] a theoretical attack against the DH key exchange protocol: an adversary could send an excessive number of public keys, which may simply be random numbers, so the victims are forced to perform a large number of modular exponentiations in order to compute the shared DH secret values, causing computational problems on the server side.

Modern cryptographic protocols support the ephemeral version of the DH key exchange which involves two modular exponentiation calculations on the server side. In most cases, an attacker can force modern protocols to perform modular exponentiation before calculating the shared secret key, namely for calculating the public key. To achieve this, one only needs to transmit an initial cryptographic handshake message. Exploiting this observation, the likelihood of a successful denial of service (DoS) attack against the ephemeral DH key exchange can be significantly increased. This small modification compared to the original idea discussed in 2002, makes this attack surprisingly easy to execute. It has turned out that even the most current servers using DH key exchange are susceptible without proper mitigation. Despite 20 years having passed since this issue was mentioned, no publicly known exploit has been developed.

A. D(HE)at ANNOUNCEMENT

In November 2021 we announced the server-side modular exponential attack, called D(HE)at attack with the assigned CVE-2002-20001 number. This attack is designed to target all variants of the DH key exchange protocol, including the ephemeral version. It is important to note that despite the D(HE)at attack being formally published in 2021, it was assigned a CVE number 2002 due to the fact that the concept of the attack for the traditional DH key exchange was already discussed in 2002. The following is the official description of the attack, as listed in the NIST NVD database: *"Diffie–Hellman Key Agreement Protocol allows remote attackers (from the client side) to send arbitrary numbers that are actually not public keys and trigger expensive server-side DHE modular-exponentiation calculations, aka a D(HE)at attack. The client needs very little CPU resources and network bandwidth. The attack may be more disruptive in cases where a client can require a server to select its largest supported key size"*.

B. IMPACT

The impact of the vulnerability is high with an assigned 7.5 CVSS (Common Vulnerability Scoring System) score. The high score is a result of various factors. Notably, the attack does not require user interaction and can be carried out remotely through the network. Additionally, the attack has a low level of complexity. It is crucial to emphasize that this specific attack does not compromise confidentiality (C), integrity (I), or scope (S), but it does have a significant impact on availability (A). The CVE-2002-20001 CVSS 3.1 radar diagram can be seen in Figure 2.

C. ATTACK DESCRIPTION.

The pseudo code and the schematic diagram illustrating the D(HE)at attack can be seen in Algorithm 3 and Figure 3. The attack deviates from the standard DH protocol right from the start, with the client stating its sole support for DHE (Algorithm 3, line 2). The malicious client can force the server to generate its public key (Algorithm 3, line 4). To carry out the attack, the attacker simply has to generate a random

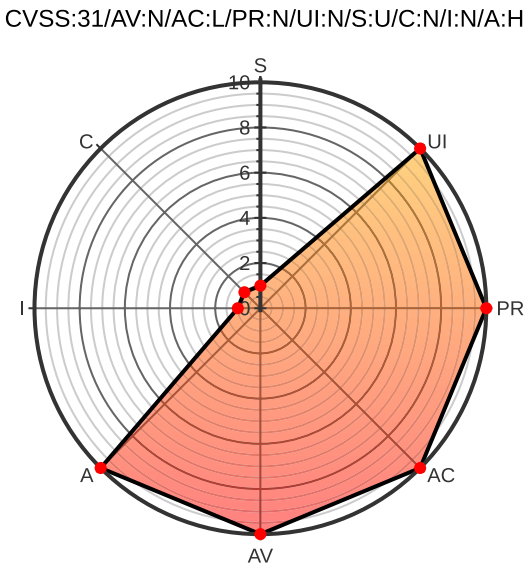

FIGURE 2. CVE-2002-20001 radar diagram.

number $A \in \mathbb{Z}_p^*$ (Algorithm 3, line 6) once and transmit it to the victim server, pretending that it was calculated as the legitimate public key. Applying this technique the malicious client can skip the expensive modular exponentiation as it is guaranteed that the result of modular exponentiation is less than the modulus p, so it is enough to pick an arbitrary number $A \in \{1, \ldots, p-1\}$.

Algorithm 3 The D(HE)at Attack

Group parameters: p prime and g generator.

1: **procedure** DHEAT()
2: **Client sends that it only supports DHE**
3: *SERVER* picks a random $b \in \mathbb{Z}_p^*$
4: *SERVER* computes $B \equiv g^b \pmod{p}$
5: *SERVER* sends (p, g) and B to *CLIENT*
6: **CLIENT picks a random** $A \in \mathbb{Z}_p^*$
7: *CLIENT* sends A to *SERVER*
8: *SERVER* computes $k \equiv A^b \pmod{p}$
9: **CLIENT skips computing the shared** k **key**
10: **return** k invalid shared secret value
11: **end procedure**

Thanks to DLP, the targeted server is unable to distinguish between a computed public key and the arbitrary random number selected by the attacker and hence accepts the random number as a legitimate public key. The server computes the fake shared key $k \equiv A^b \pmod{p}$ (Algorithm 3, line 8) and the client skips the expensive modular exponentation once again (Algorithm 3, line 9). With this mechanism, a malicious client can force the server to perform modular exponentiation twice, while avoiding the need to perform any computationally intensive calculations on the client side. While calculating modular exponentiation has a time complexity of $\mathcal{O}(\log b)$, meaning it scales logarithmically with the size of the exponent b, generating a random number

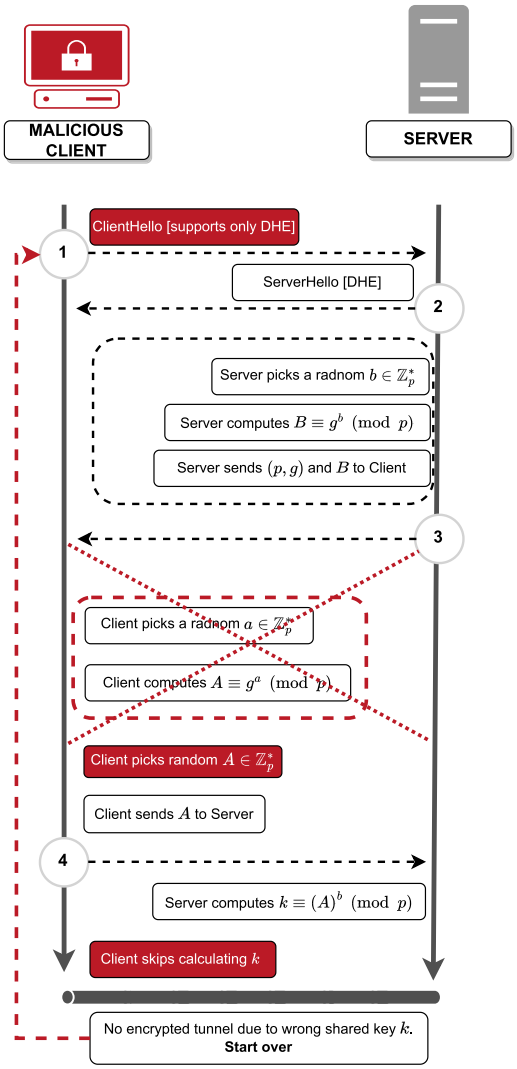

FIGURE 3. D(HE)at attack concept.

has a constant time complexity of $\mathcal{O}(1)$. The fact that generating a random number has a constant time regardless of the input size makes it advantageous for potential attackers, as it allows for efficient execution of the D(HE)at attack.

D. PROOF OF CONCEPT

Even though the D(HE)at attack has significant consequences, it was considered a performance issue by most vendors due to its fundamental nature. To underscore the significance of this matter, we created a PoC code (D(HE)ater [50]) for the D(HE)at attack that could operate effectively in almost every situation for different protocols, with the aim of convincing the impacted vendors. By openly sharing the code, researchers can work together to improve the security of cryptographic libraries and reduce the risks posed by this vulnerability.

In the following section, we will illustrate how the D(HE)at attack can be easily carried out against the latest operating systems (e.g. Ubuntu 22.04 LTS or Ubuntu 23.04) with all

the latest security patches installed, assuming no additional countermeasures are in place (e.g. Fail2ban, WAF). We would like to emphasize that although we will demonstrate the attack on the most recent Ubuntu servers equipped with the latest OpenSSL, it is important to note that the attack is not specific to these systems. It is a generic issue in the DH key exchange and can be exploited on various operating systems with different cryptographic libraries. The examples presented in the following section were also tested on various major operating systems such as Debian 11, openSUSE Leap 15.3, Fedora, and others. This emphasizes the criticality that, the vulnerability of the DH key exchange to modular exponential attacks is not a theoretical concern but has significant implications in practical real-life situations.

E. DEMONSTRATION OF THE D(HE)at ATTACK

Before we begin examining and comparing the various cryptographic libraries, we will showcase the effectiveness of the D(HE)at attack by testing it on different types of protocols, namely TLS 1.3, and SSH 2.0. The tests are highly illustrative, and in every instance, we managed to achieve 100% CPU utilization on the server side without any resource-intensive computations on the client side.

1) 2 CPU CORES INSTANCE – TLS 1.3

In our first example, a Digital Ocean Regular instance was set up with two Intel CPU cores (Premium NVMe SSD Droplet with 4 GB RAM). We deployed Ubuntu 22.10, the latest available instance at the time of writing on the Digital Ocean cloud service. Following this, we manually updated the droplet to the latest version, Ubuntu 23.04 (Lunar Lobster). Apache2 (2.4.57) web server along with the latest OpenSSL 3.0.8 were installed. All patches and security updates were applied to the server. Within the Apache2 configuration file, we have activated only TLS 1.2 and TLS 1.3 with the latest cipher suites recommendations. In the configuration file, only eight strong cipher suites were activated:

```
ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-RSA-AES128-GCM-SHA256
ECDHE-ECDSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-ECDSA-CHACHA20-POLY1305
ECDHE-RSA-CHACHA20-POLY1305
DHE-RSA-AES128-GCM-SHA256
DHE-RSA-AES256-GCM-SHA384
```

LISTING 6: Enabled cipher suites in our test server.

The remote server will become inaccessible when the command `dheat.py -thread-num 64 -protocol tls <IP>` is executed. The proof of concept tool automatically selected the largest DH public key size available in TLS 1.3, which in our case was 8192 bits. Within 15-20 seconds, the load average reaches 150, and the CPU utilization hits 100%, as can be seen in Figure 4.

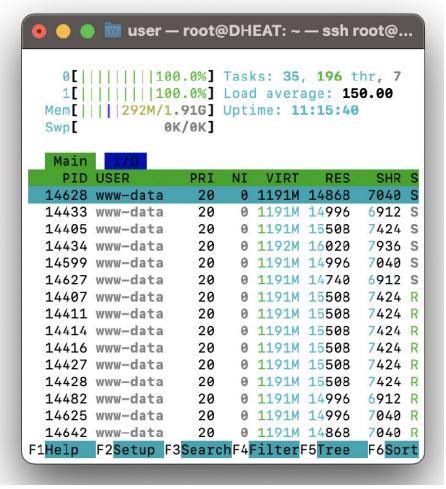

FIGURE 4. D(HE)at attack against the TLS 1.3 protocol.

The server becomes inaccessible during the attack and is unable to handle additional requests. In case someone attempts to download the server's main web page using curl, the following error is encountered `curl: (28) Failed to connect to <IP> port 443.`

2) 2 CPU CORES INSTANCE – SSH 2.0

Using the same instance (Ubuntu 23.04, OpenSSL 3.0.8) we carried out the same attack against the SSH protocol. The connection that was established during the attack utilized the SSH 2.0 protocol and employed the `diffie-hellman-group18-sha512` algorithm, which means an 8192-bit key size similar to the previous attack scenario. Within 35-40 seconds, both CPU cores once again attained their peak capacity. Once the load average exceeds 50, the server becomes incapable of accepting the majority of further SSH connections, resulting in dropping approximately 6-7 out of every 10 connections: `kex_exchange_identification: Connection closed by remote host.` In these two examples, we used the most popular Digital Ocean instance with 2 CPU cores. The question naturally arises: Can increasing the number of CPU cores prevent or mitigate the impact of this attack? The answer is no, which we will illustrate in our upcoming example.

3) 32 CPU CORES INSTANCE – TLS 1.3

We successfully executed an attack on the most recent Ubuntu 23.04 in the previous two examples, which had OpenSSL 3.0.8 installed. It is important to note that the majority of users choose the long-term support (LTS) version of Ubuntu. The reason behind this preference is that LTS releases offer extended stability, security, and support compared to normal releases. Therefore, to demonstrate a more realistic scenario we will now attempt our attack on the most recent Ubuntu 22.04 LTS (Jammy Jellyfish) with a significantly more powerful computer, specifically an AMD

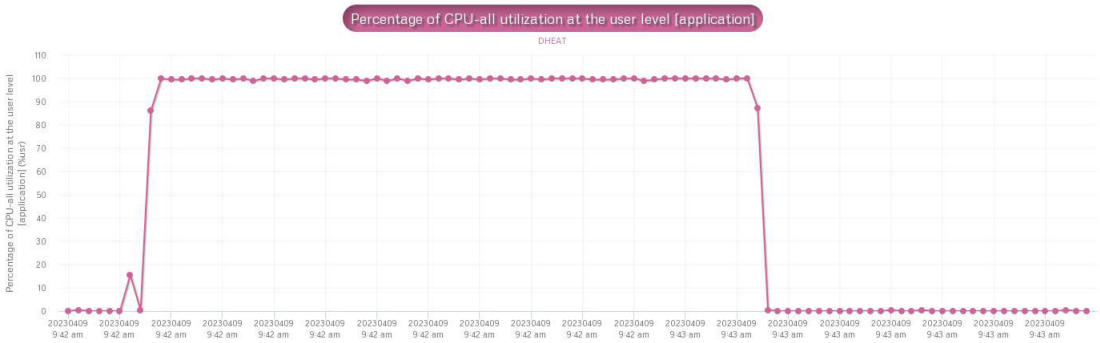

FIGURE 5. 100% CPU usage during a D(HE)atattack against an AMD Ryzen Threadripper PRO 3995WX with 32 CPU cores.

Ryzen Threadripper PRO 3995WX with 32 CPU cores. This server is equipped with a pre-installed OpenSSL version 3.0.2 (which is the default OpenSSL version in Ubuntu 22.04). In this example, our focus is once again on the HTTPS protocol. In order to remove any potential bias towards a specific service provider, we decided to switch from Digital Ocean to a private data center situated in the Middle East region.

Even when attacking a more powerful server with additional CPU cores, the result remains unchanged. All 32 CPU cores reach 100% utilization, as depicted in Figure 5. When the load average reaches $80 - 90$ the server experiences significant slowdowns, making it unable to handle normal requests effectively.

The attack was successful, howeverer, a notable difference can be observed. We attained 100% utilization across all 32 CPU cores in merely 5 seconds, employing 32 threads with the D(HE)at tool, as opposed to the prior example where 64 threads were used and peak CPU utilization was reached after 20-40 seconds. How did we manage to overwhelm a significantly greater server capacity, while using fewer resources and less time from the client-side perspective?

F. LONG EXPONENTS (CVE-2022-40735)

The answer to the previous question can be discovered by examining the differences between OpenSSL 3.0.2 (Ubuntu 22.04 LTS) and OpenSSL 3.0.8 (Ubuntu 23.04). In the more recent OpenSSL version (3.0.8), a patch has been implemented to mitigate the D(HE)at risk associated with using large private keys. In OpenSSL 3.0.2, the D(HE)at impact is considerably more visible, as the DH key exchange protocol is using unnecessarily large private keys (exponents) making the D(HE)at attack significantly more feasible.

Some cryptographic libraries implement the Diffie–Hellman key exchange by using long exponents arguably making modular exponentiations unnecessarily expensive. Appropriately short exponents can be used when there are adequate subgroup constraints [80], and these short exponents can lead to less expensive calculations than long exponents with the same security strength. This implementation issue enhances the effect of the D(HE)at attack, especially when larger key sizes are enabled in the server configuration or the private exponent size is not configurable. It means the D(HE)at attack can be more effective if the private exponent size is significantly larger on the server side.

This is exactly the case in our third, 32 CPU cores example. In OpenSSL 3.0.2, an unnecessarily large private key exponent (8192 bits) is associated with the 8192-bit prime modulus. As per the international standards, including the NIST recommendations (c.f. Table 2), for an 8192-bit p modulus, a roughly 380-400 bits private exponent suffices to achieve adequate security. Utilizing a key larger than 400 bits does not yield any additional security but can lead to a significant performance dropdown (as shown in Section II-D2). The CVE-2022-40735 identifier was assigned for this type of implementation flaw and was publicly disclosed in November 2022.

In OpenSSL 3.0.6 the ‘‘long exponent’’ issue (CVE-2022-40735) is already mitigated, and for an 8192-bit modulus, only a 400-bit associated private key size is presented. This implies that significantly more requests are required from the client to execute the D(HE)at attack effectively. If both CVE-2002-20001 and CVE-2022-40735 vulnerabilities are present on a vulnerable server, especially if large group parameters are enabled, a malicious client can completely exhaust the server’s resources with minimal effort resulting in a highly efficient DoS attack.

As discussed at the beginning of this section, the latest Ubuntu 22.04 LTS version which is using OpenSSL 3.0.2 still contains the CVE-2022-40735 vulnerability. However, this vulnerability has been resolved in Ubuntu 23.04 with the use of OpenSSL 3.0.8. As a result, only the original D(HE)at attack (CVE-2002-20001) can be used against Ubuntu 23.04. Despite the absence of the long exponent problem (CVE-2022-40735) in Ubuntu 23.04, a malicious client can still exhaust server resources from a normal laptop (e.g. 2017 MacBook Pro, 16GB RAM) as we have shown in the first two examples. Since different applications and servers employ various cryptographic libraries, the question arises as to which library is affected and which one should be used by

system administrators. Answering this question is a complex task, as it requires a thorough understanding of the unique requirements of each application and we need to explore the default settings of different cryptographic libraries.

In the following section, we present a thorough assessment of different cryptographic libraries. Our analysis includes a comprehensive comparison of the default settings for cryptographic libraries, as well as the default sizes of private keys and group parameters used by these libraries.

IV. ANALYSIS

In this section, we delve into a comprehensive methodology that is essential for accurately assessing various cryptographic libraries. We also emphasize the significance of assessing the speed of cryptographic libraries, as opposed to focusing solely on the speed of the application server.

A. METHODOLOGY

Evaluating the DH key exchange at the application level would yield unrealistic outcomes. This is because application servers like Apache, Lighttpd, and NGINX engage in various additional tasks, such as socket handling and thread creation, during the cryptographic handshake process, making the measurement results impossible to compare. A possible solution could involve the utilization of a specific application server. However, it is worth noting that there is no single application server that supports all cryptographic libraries and all the investigated protocols. Therefore, we evaluated cryptographic libraries, as they operate at the lowest level of DHE key exchange. If a cryptographic library is vulnerable to a D(HE)at attack, all application servers using that library are also at risk. The result of measuring the cryptographic library represents a theoretical maximum performance that can be reached by any application server utilizing a specific cryptographic library.

1) DH KEY EXCHANGE SUPPORT

In the previous section, we illustrated the practical usage of the D(HE)at attack. However, a fundamental question naturally emerges: What is the current status of DH key exchange support and key sizes among servers? Does the D(HE)at attack effectively target the majority of servers present on the internet? We have conducted an investigation into the two most widely used protocols that offer DH key exchange: TLS and SSH.

We found that 55% of the top 1 million [32] web servers utilize the DH key exchange protocol during the TLS negotiations (see Figure 1). According to our measurements, web servers that utilize DH key exchange typically do not provide compatibility with key sizes exceeding 2048 bits in the majority of cases (see Figure 6).

The reason for this is that, before TLS version 1.2, the negotiation of DH parameters was not a part of the TLS standard; it was introduced as an extension in RFC7919 [17]. Moreover, the popular OpenSSL lacks support for the necessary extension to negotiate DH parameters prior to TLS

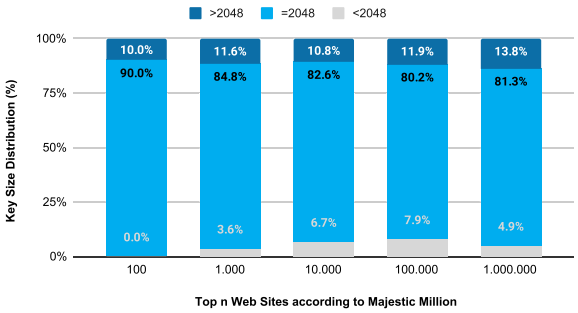

FIGURE 6. Diffie–Hellman key size support in the Top 1 million domains.

1.2. Only its latest major version (3.0) supports DH group parameter negotiation in TLS version 1.3. With the increasing popularity of TLS 1.3, it is reasonable to anticipate greater support for larger keys, making D(HE)at more efficient. This is particularly relevant because OpenSSL (3.0) utilizes the largest possible (8192 bit) DH group parameter by default. This fact gains more relevance when considering that the configurations of most application servers are dependent on the cryptographic library settings.

Beyond that, the widespread usage of the SSH protocol on the internet can be assessed in a similar manner. According to the data gathered from Shodan [73], we found (see Figure 7) that 87% of servers worldwide offer support for DHE within the SSH protocol, 65% offer the largest key size defined in the related standard [5], and 82% support group exchange [15] that also makes possible negotiation of larger key sizes.

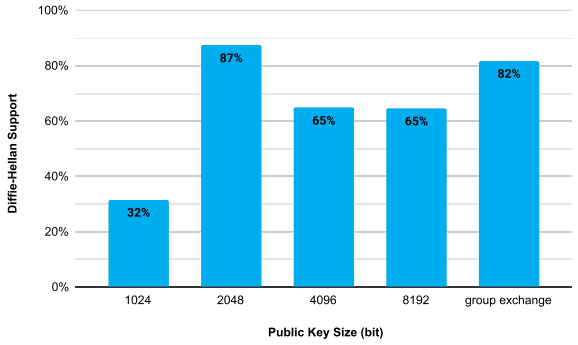

FIGURE 7. Diffie–Hellman key size support in SSH according to Shodan.

2) EVALUATED ARCHITECTURES

Throughout the evaluation process, we involved four distinct types of CPUs, ranging from low to high-end performance. These CPUs are the following:

- ARM v7 Processor rev 4 (v7l)
- Digital Ocean Regular
- Intel Atom C3558 CPU @ 2.2GHz
- Intel Xeon E-2224 CPU @ 3.40GHz

In all our measurements, we utilized a single core and a single thread from each CPU. The rationale behind this approach was that various application servers and cryptographic libraries leverage cores differently, and we

aimed to generate comparable results. Moreover, in modern architectures, these CPUs are frequently used in virtual environments, allowing users to select the number of cores and threads they wish to use. With a single core and thread configuration, the measurements presented in this study allow for easy replication and adjustment according to the desired number of cores and threads.

3) EVALUATED LIBRARIES

We have conducted research on all the major cryptographic libraries across the specified CPU types including OpenSSL, BoringSSL, LibreSSL, GnuTLS, NSS, Mbed TLS, OpenJDK and WolfSSL. The analyzed libraries and their corresponding version can be seen in Table 3.

TABLE 3. Compared cryptographic libraries.

Library	Evaluated Version
OpenSSL 1.0	1.0.2u
OpenSSL 1.1.1	1.1.1t
OpenSSL 3.0	3.0.8
BoringSSL	3.8
LibreSSL	3.7.2
GnuTLS	3.7.9
NSS	3.88
Mbed TLS (PolarSSL)	3.4.0
OpenJDK	17.0.6
Oracle JDK	17.0.6
WolfSSL	5.6.0

Since there were significant changes in the Diffie–Hellman implementation and parameters used between versions 1.0, 1.1.1, and 3.0 of OpenSSL, we made sure to include these three major versions in our performance measurements. By evaluating the performance of each version under different scenarios, we were able to gain a better understanding of the impact that these changes had on the overall performance of the OpenSSL library.

The performance of the Diffie–Hellman key exchange strongly depends on the used private-public key sizes, meaning that the effectiveness of the D(HE)at attack also strongly depends on that. Thus, our first comparison aims to evaluate the effective key generation speed (modular exponentiation speed) in the aforementioned libraries using different p modulus and different private key sizes.

B. KEY GENERATION PERFORMANCE

Different cryptographic libraries come with varying default settings, offering support for different maximum modulus sizes. Furthermore, there is a significant difference in the speed of modular exponentiation across various library implementations, which is a crucial factor in determining the public key generation speed. Interestingly, this difference was even observed among different versions of the same cryptographic library, emphasizing the importance of carefully selecting the appropriate library version for specific applications to achieve optimal performance. In the first step of our investigation, we focused on evaluating the performance of the DH key generation. To ensure consistent and comparable results, we measured key generation speeds using the same parameters, specifically a 2048-bit public key and a 232-bit exponent size. Figure 8 illustrates the DH key generation speed of various cryptographic libraries relative to PARI/GP (100%) on the Intel Xeon E-2224 CPU @ 3.40GHz platform.

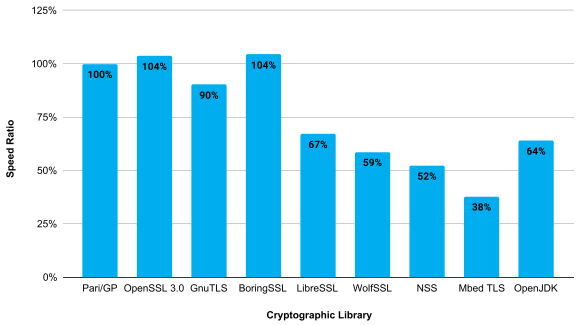

FIGURE 8. DH key generation speed of cryptographic libraries compared to PARI/GP on Intel Xeon E-2224 CPU @ 3.40GHz .

Just for clarification, when we are discussing key generation in the context of the DH key exchange protocol, we are referring to the computation of the public key, represented as $A \equiv g^a \pmod{p}$. To establish a reference point, throughout this paper, we used PARI/GP version 2.15.2 and measured the performance of all other cryptographic libraries against it.

By comparing the modular exponentiation (key generation) speed of each library to PARI/GP, we were able to identify significant differences in performance and determine which libraries performed best under different scenarios. Only two (OpenSSL, BoringSSL) of the investigated cryptographic libraries can exceed the performance of PARI/GP. GnuTLS is close to the result of PARI/GP, but all the others significantly fall short of the performance that can be considered optimal.

It is important to acknowledge that the significant variations among different libraries primarily arise from their requirement to be compatible with a wide range of platforms and devices. Although, the performance differences are significant and establish the need for optimization, in real-life scenarios the DH key generation parameters – public key and exponent sizes – and their default matter much more than the pure performance.

1) LONG EXPONENT ATTACK

A significant difference can be seen in performance when comparing the key generation speed, which is heavily dependent on the size of the used exponent in the modular exponentiation. For instance, when using OpenSSL 3.0.5 with a 3072-bit modulus and a private key of the same size, it is possible to generate 208 public keys per second per CPU thread on Intel Xeon E-2224 CPU @ 3.40GHz . However, when the private key size is reduced in OpenSSL 3.0.6 to 272 bits, by NIST recommendations, the public key generation speed increases to 1811 per second, which is a

770% speed improvement. Figure 9 illustrates the significant difference in speed between a small and a large private key (232 vs. 3072 bits) using various libraries on Intel Xeon E-2224 CPU @ 3.40GHz .

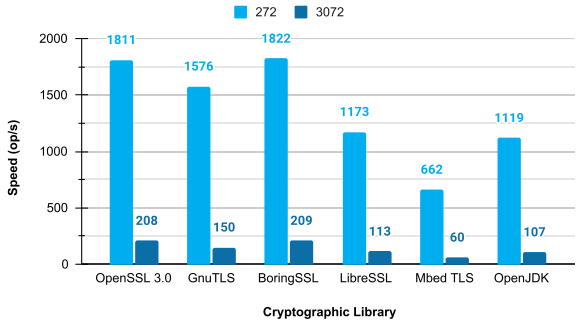

FIGURE 9. DH key generation speed using small and long exponents on Intel Xeon E-2224 CPU @ 3.40GHz .

Given this significant difference in performance speed, it becomes essential to investigate the private key sizes employed by cryptographic libraries. To select the most suitable library for diverse applications and server requirements, we aim to conduct a more detailed comparison involving different key sizes. The speed of public key generation for small exponent sizes (232-416 bits) is presented in Table 6 and Figure 10. For long exponent sizes (2048-8192 bits) the key generation speed can be seen in Table 7 and Figure 11. Both tables include results for all cryptographic libraries tested on the four CPU instances. One can immediately see the performance difference between the small and large private keys.

2) DEFAULT PRIVATE KEY SIZE

Typically, users do not modify the default configurations of cryptographic libraries or make changes to the source code before compilation. As a result, the size of the private keys used in the DH key exchange is determined by the default configurations of the respective cryptographic libraries. For example, each version of OpenSSL 1.0/1.1.1 and 3.0 until 3.0.5 are vulnerable to the "long exponent" issue (CVE-2022-40735), meaning that large private keys were in use by default. This issue was fixed in 3.0 series starting with version 3.0.6, where each public key size is paired with a smaller private key according to the NIST recommendation. The only exception is when custom (different from the ones defined in RFC 7919 [17] or RFC 3526 [29]) parameters (p, g) are set using a DH parameter file, and the optional private exponent value is not set. In that case "long exponent" issue still exists in OpenSSL 3.0 versions utilizing DH key exchange up to the 1.2 version of TLS.

To mitigate the attack surface of the "long exponent" issue, it is crucial to utilize a cryptographic library that uses short exponents. We have collected and documented (default) private key sizes of each investigated cryptographic library.

Table 4 demonstrates that the "long exponent" issue is not as unlikely as one might assume considering that the issue has been known for more than twenty years [68] and the related RFCs formulate recommendations for the applicable private exponent sizes (c.f. Table 2). In part of this study, all cryptographic library vendors were warned about this issue. As a consequence, the latest versions of OpenJDK, Oracle JDK, and OpenSSL use small exponents. One can see that GnuTLS, NSS, and WolfSSL have never been affected in the "long exponent" issue. Detailed information about the key generation speeds of the investigated cryptographic libraries can be found in Table 6 and Table 7 which illustrate a noticeable performance decrease across different key sizes and libraries.

TABLE 4. Default TLS public key sizes and private exponents.

Library \ Modulus		2048	3072	4096	6144	8192
Name	Version	Default Private Exponent				
OpenSSL	$\leq$ 1.1.1 [62]	2048	3072	4096	6144	8192
OpenSSL	$\leq$ 3.0.5 [64]	2048	3072	4096	6144	8192
OpenSSL	$\geq$ 3.0.6 [63]	225	275	325	375	400
BoringSSL*	3.8	n/a	n/a	n/a	n/a	n/a
LibreSSL*	3.7.2	n/a	n/a	n/a	n/a	n/a
GnuTLS	3.7.9 [52]	256	276	336	376	512
NSS	$\leq$ 3.13 [55]	160	160	160	160	160
NSS	$\geq$ 3.14 [56]	224	256	256	256	384
Mbed TLS	3.4.0 [54]	2048	3072	4096	6144	8192
OpenJDK	$\leq$ 17.0.5 [58]	1024	1536	2048	3072	4096
OpenJDK	$\geq$ 17.0.6 [59]	224	256	304	352	400
Oracle JDK	$\leq$ 17.0.5	1024	1536	2048	3072	4096
Oracle JDK	$\geq$ 17.0.6	224	256	304	352	400
WolfSSL	5.6.0 [66]	232	272	312	368	416

* no finite field Diffie–Hellman key exchange support

3) PUBLIC MODULUS SIZE

Another crucial factor influencing key generation speed is the public modulus size. For instance, a significant difference is evident between an 8192-bit modulus and a 3072-bit modulus, even when the appropriate private key size is utilized. In order to attain 128-bit security, the NIST recommends using a 3072-bit prime modulus and a minimum private key size of 256 bits (see Table 1). If a client can force the server to use the largest supported group parameter during the DH key exchange parameter negotiation, such as an 8192-bit modulus, a substantial performance decline becomes visible between the two moduli. As an example, this performance difference can reach 967% (1811 op/sec vs 187.2 op/sec) on Intel Xeon E-2224 CPU @ 3.40GHz using OpenSSL 3.0, as shown in Table 6 and Table 7.

4) ALGORITHM AND PARAMETER NEGOTIATION

At this point, we need to clarify an important question. Is it actually possible for the client to force the server to utilize the larger key sizes, thus increasing the impact of D(HE)at attack? It primarily depends on whether DH group parameter negotiation is defined in a cryptographic protocol. If so, how widespread is the support of parameter negotiation among the most popular cryptographic library implementations, and which parameters are offered by default?

a: CRYPTOGRAPHIC PROTOCOLS

All widely used cryptographic protocols have DH parameter negotiation support (see Table 5).

TABLE 5. Cryptographic protocol DH parameter negotiation support.

Protocol	Parameter Negotiation	
	Not Supported	Supported
TLS	1.0 - 1.2 [9]	1.2, with extension [17] 1.3 [70]
SSH		2.0 [16] [84]
IKE		v1 [21] v2 [25]

The only exception is the TLS protocol prior to its version 1.2 which supports the negotiation only with an extension.

In the Diffie–Hellman key exchange protocol, there are two methods to agree on group parameters (p, g) during the handshake. One such method involves referencing the group parameters known to both parties, such as those defined in RFC 3526 [29], RFC 5114 [26], and RFC 7919 [17]. Most protocols employ this approach (see Table 8) as it can decrease the bandwidth requirement during the cryptographic handshake, especially in the case of larger key sizes (e.g. 8192-bit). The other method involves directly transmitting group parameters, which has the advantage that any parameters can be used independently whether they are standardized or not. Irrespective of whether the Diffie–Hellman parameters are referenced through an RFC document or transmitted directly during a cryptographic handshake, it is important to note that only the group parameters (p, g) are negotiated between the involved parties. The size of the private exponents (a, b) is determined independently by each party without any reliance on the other.

The cryptographic protocols definition can either specify that a server should respect the other client's algorithm preference or disregard it. If the client preference is honored, the attacker can order the algorithms based on their requirements, giving priority to the most resource-intensive ones. On the other hand, if the server disregards the client's algorithm preference, the attacker is required to examine the victim's service to identify the supported algorithms. In this case, the malicious client can always claim that he can only negotiate the most resource-intensive algorithm applicable to the victim. It means that the only prerequisite of the D(HE)at attack is that the server is configured to allow Diffie–Hellman key exchange. The attacker's fundamental interest is to enforce the largest key size, as the resource requirement of the modular exponentiation highly depends on the size of the public modulus (p). In certain cryptographic protocols, negotiation is not feasible (see Table 5) as the server offers the group parameters and the client does not influence it. However in many cases, such as with TLS 1.2 implementing RFC 7919, TLS 1.3, or SSH 2.0 employing Diffie–Hellman support, the client offers a list of preferred cipher suites with the corresponding parameters, enabling an attacker to force the victim to use the largest enabled key size. However, cryptographic protocols have various ways to negotiate DH group parameters (p, g) during the Diffie–Hellman key exchange part of the cryptographic handshake, but only whether they support parameter negotiation affects the effectiveness of the D(HE)at attack, regardless of how they support it.

b: LIBRARY IMPLEMENTATIONS

The overwhelming majority of cryptographic libraries that implement the Diffie–Hellman key exchange also implement group parameter negotiation in TLS (see Table 9). The one exception is Mbed TLS which does not implement parameter negotiation in any TLS versions. The situation was the same in the case of the last major version (1.1.1) of OpenSSL, but the latest major version (3.0) released with parameter negotiation support, but only in the 1.3 version of TLS. As a consequence, it is practically feasible for an attacker to force larger DH key sizes during the D(HE)at attack making the negotiable parameter enabled by default in the cryptographic library particularly important.

All the investigated cryptographic libraries that support DH parameter negotiation in TLS also enable the largest key size (8192 bits) in the related standard by default. It means that if the implementation or the configuration of the application server does not override this default value the attacker could enforce the server to perform the most resource-intensive DH key exchange variant. The only exception is WolfSSL as shown in Table 10, which disables the key sizes larger than 2048 bits in the library implementation by default, so the application server cannot enable it.

C. RESOURCE REQUIREMENTS FOR D(HE)at

1) THROUGHPUT

The necessary throughput for a successful D(HE)at attack is determined by several factors, including the maximum Diffie-Hellman public key size supported by the server's configuration, the private key sizes employed by the cryptographic library or the application server, and the computational capacity of the targeted machine's CPU. As a part of the DH key exchange process, the computationally intensive modular exponentiation is carried out twice. To determine the number of requests required to fully utilize a single CPU thread, one can consider half of the measured DH key generation speed values. Therefore, to calculate the total number of requests needed to fully load each core of all the CPUs within a machine, one should multiply this value by the total number of CPU cores and threads available.

If an attacker can send the necessary amount of requests successfully – meaning that the requests are received and processed by the server – it is guaranteed that it causes 100% load on each CPU core, as only the DH key generations are enough to achieve that CPU load. In practice, the amount of successfully sent requests is lower as an attacked server usually performs several other operations (such as thread

TABLE 6. DHE public key generation speed - small exponent 232-416 bits.

Key Size (bit)		Number of op/sec for 1 CPU thread for various libraries										
Public Key	Private Key	Pari-GP	Open-SSL 1.0	Open-SSL 1.1	Open-SSL 3.0	Gnu-TLS	Boring-SSL	Libre-SSL	Wolf-SSL	NSS	Mbed TLS	Open-JDK
Intel Xeon E-2224 CPU @ 3.40GHz												
2048	**232**	4043	4494	4389	4393	3824	4470	2930	2742	2464	1623	2738
3072	**272**	1746	1823	1817	1811	1576	1822	1173	1026	913.4	661.6	1119
4096	**312**	961.5	1049	1004	1001	810.7	1053	613.3	583.6	547.7	335.4	551.9
6144	**368**	404.7	404.2	402.4	384.6	394.5	404.6	237.0	221.2	259.6	128.8	214.3
8192	**416**	239.1	187.6	196.1	187.2	223.1	187.9	119.0	105.5	101.4	65.29	105.0
Digital Ocean Regular												
2048	**232**	1908	1407	1398	1314	1610	1344	1419	1467	1184	825.2	1385
3072	**272**	776.0	530.9	522.3	527.0	721.2	541.3	498.4	502.3	484.3	311.8	534.5
4096	**312**	427.2	308.9	309.2	297.5	366.6	318.7	287.6	293.5	283.6	156.0	270.8
6144	**368**	179.9	113.4	117.7	114.7	171.7	108.2	107.8	106.1	133.8	60.93	112.1
8192	**416**	108.0	56.64	56.03	57.41	102.2	58.43	55.76	53.25	51.2	30.99	55.3
Intel Atom C3558 CPU @ 2.2GHz												
2048	**232**	1182	757.5	748.6	746.3	1073	765.0	751.2	679.2	661.8	452.9	679.7
3072	**272**	490.5	288.2	288.3	288.0	457.1	289.9	301.7	257.0	247.9	186.1	266.7
4096	**312**	265.9	164.8	164.8	164.8	236.6	165.5	159.8	144.7	146.0	94.96	134.0
6144	**368**	111.5	62.68	62.7	62.67	99.49	62.82	62.04	54.86	68.31	37.17	52.84
8192	**416**	59.0	30.71	30.72	30.71	53.06	30.77	31.27	26.92	26.56	18.87	26.8
ARM v7 Processor rev 4 (v7l)												
2048	**232**	286.6	147.2	244.9	244.8	224.7	238.1	71.01	157.8	82.83	119.8	184.4
3072	**272**	117.5	51.47	90.49	90.59	94.75	89.94	27.6	61.32	33.43	48.27	74.5
4096	**312**	62.96	33.26	51.77	51.9	52.69	51.65	13.75	35.11	18.78	24.27	37.0
6144	**368**	26.86	11.74	19.73	19.74	23.91	19.69	5.29	12.81	8.61	9.09	14.71
8192	**416**	14.7	6.65	9.66	9.69	13.39	9.66	2.59	6.39	3.32	4.73	7.22

TABLE 7. DHE public key generation speed - large exponent 2048-8192 bits.

Key Size (bit)		Number of op/sec for 1 CPU thread for various libraries										
Public Key	Private Key	Pari-GP	Open-SSL 1.0	Open-SSL 1.1	Open-SSL 3.0	Gnu-TLS	Boring-SSL	Libre-SSL	Wolf-SSL	NSS	Mbed TLS	Open-JDK
Intel Xeon E-2224 CPU @ 3.40GHz												
2048	**2048**	441.0	660.0	631.8	661.2	463.6	664.7	355.3	n/a	n/a	190.2	339.3
3072	**3072**	149.1	207.9	208.2	208.1	149.6	208.5	113.2	n/a	n/a	60.36	107.0
4096	**4096**	70.62	89.84	89.99	85.48	65.73	90.18	49.52	n/a	n/a	26.21	44.94
6144	**6144**	23.36	25.86	25.89	25.89	25.28	27.23	15.18	n/a	n/a	7.88	13.38
8192	**8192**	11.79	11.49	11.47	11.48	12.17	10.95	6.49	n/a	n/a	3.4	5.46
Digital Ocean Regular												
2048	**2048**	193.1	207.0	203.3	187.2	190.1	192.9	157.6	n/a	n/a	88.59	152.3
3072	**3072**	62.45	57.73	58.26	58.66	67.7	59.96	52.77	n/a	n/a	28.75	51.17
4096	**4096**	29.65	24.95	24.95	24.6	28.43	21.82	21.89	n/a	n/a	12.33	22.24
6144	**6144**	9.86	7.24	7.42	7.54	10.76	7.42	6.8	n/a	n/a	3.64	6.49
8192	**8192**	5.09	3.02	3.03	3.26	4.93	3.19	2.95	n/a	n/a	1.55	2.73
Intel Atom C3558 CPU @ 2.2GHz												
2048	**2048**	128.1	106.6	106.7	106.7	130.1	106.9	90.92	n/a	n/a	53.12	83.18
3072	**3072**	41.12	32.42	32.41	32.41	43.36	32.43	28.98	n/a	n/a	16.95	25.25
4096	**4096**	19.38	13.88	13.85	13.88	19.23	13.87	12.74	n/a	n/a	7.4	10.77
6144	**6144**	6.42	4.18	4.18	4.18	6.38	4.18	3.92	n/a	n/a	2.27	3.29
8192	**8192**	2.9	1.78	1.78	1.78	2.89	1.78	1.69	n/a	n/a	0.98	1.39
ARM v7 Processor rev 4 (v7l)												
2048	**2048**	31.17	19.75	33.22	33.29	28.13	32.16	8.66	n/a	n/a	14.0	22.57
3072	**3072**	10.0	5.78	10.02	10.34	8.97	10.15	2.64	n/a	n/a	4.38	7.02
4096	**4096**	4.63	2.81	4.47	4.47	4.29	4.28	1.13	n/a	n/a	1.88	3.05
6144	**6144**	1.56	0.79	1.35	1.35	1.53	1.34	0.34	n/a	n/a	0.57	0.92
8192	**8192**	0.72	0.39	0.58	0.58	0.73	0.58	0.14	n/a	n/a	0.24	0.38

creation or memory management) than the DH key exchange part of handling a new client connection.

2) BANDWIDTH

The bandwidth requirement of the D(HE)at attack highly depends on the cryptographic protocol peculiarities. The Diffie–Hellman key exchange requires performing the modular exponentiation twice (public key generation, shared secret calculation). Still, a malicious actor may choose an attack pattern that forces performing only one modular exponentiation per connection, because it may require much smaller bandwidth or just one round-trip (see Table 11), which may mean a better cost efficiency ratio on the attacker's side.

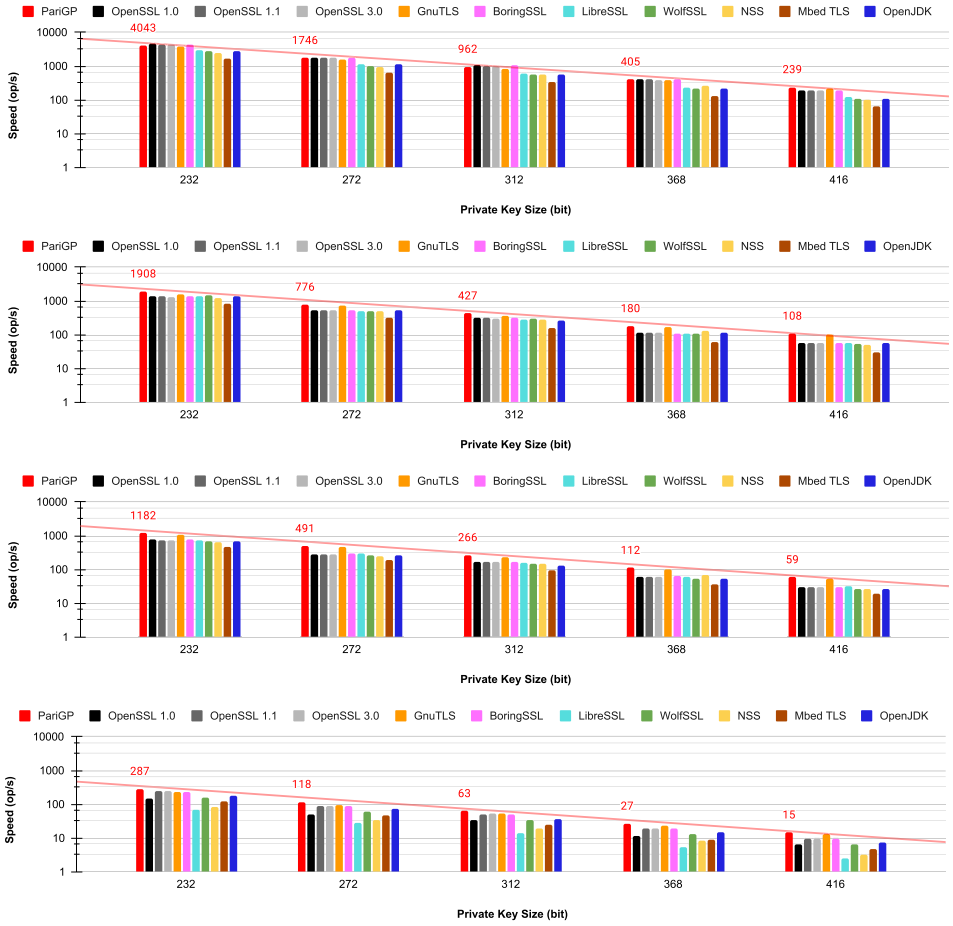

FIGURE 10. DHE public key generation speed - small exponent 232-416 bits.

TABLE 8. Cryptographic protocol DH parameter transmission method.

Protocol	Parameter Transmission Method	
	Direct	Referencing
TLS	1.0 - 1.2 [9]	1.2, with extension [17] 1.3 [70]
SSH	2.0 [15]	2.0 [15] [85]
IKE		v1 [21] v2 [25]

TABLE 9. Cryptographic library DH parameter negotiation support in TLS.

Library	TLS 1.2 (RFC 7919)	TLS 1.3
OpenSSL	no	$\geq$ 3.0 [61]
BoringSSL*	n/a	n/a
LibreSSL*	n/a	n/a
GnuTLS	$\geq$ 3.6.0 [18]	$\geq$ 3.6.0 [18]
NSS	$\geq$ 3.20 [34]	$\geq$ 3.25 [35]
Mbed TLS	no	no [78]
OpenJDK	$\geq$ 19 [46]	$\geq$ 11 [45]
Oracle JDK	$\geq$ 18 [46]	$\geq$ 11 [46]
WolfSSL	$\geq$ 5.2 [81]	$\geq$ 5.0.0 [81]

* no finite field Diffie–Hellman key exchange support

The proof-of-concept implementation of the D(HE)at attack (D(HE)ater) always waits for a cryptographic handshake message from the server that proves that the computationally demanding modular exponentiation has already been performed. On the one hand, it is not the most effective attack pattern as increases the amount of time and bandwidth necessary for a connection. On the other hand, depending on the server implementation details it may be enough to send the cryptographic handshake message that triggers modular exponentiation, and the connection can be closed immediately.

TABLE 10. Maximum public key size offered by default in cryptographic libraries during DH parameter negotiation of TLS.

Library	Maximum Prime Modulus Enabled by Default
OpenSSL	8192 [65]
BoringSSL*	n/a
LibreSSL*	n/a
GnuTLS	8192 [53]
NSS	8192 [57]
Mbed TLS**	n/a
OpenJDK	8192 [60]
Oracle JDK	8192 [60]
WolfSSL	2048 [67]

* no finite field Diffie–Hellman key exchange support
** no finite field Diffie–Hellman parameter negotiation support

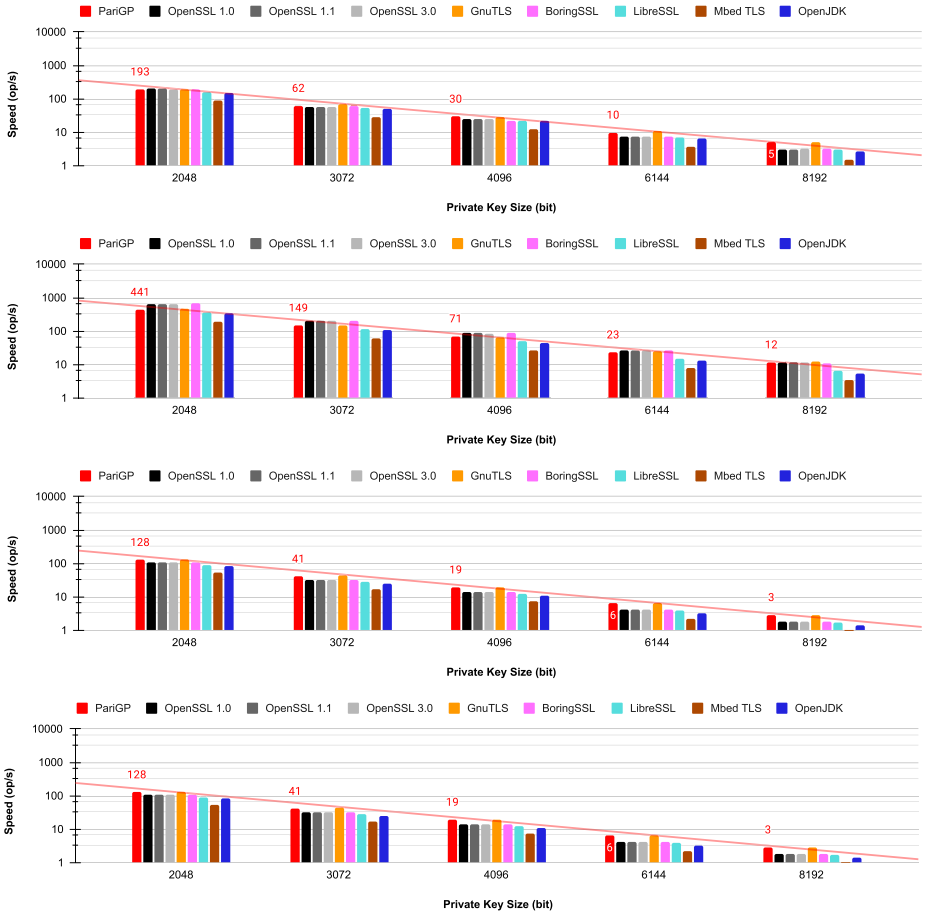

FIGURE 11. DHE public key generation speed - large exponent 2048-8192 bits.

TABLE 11. Cryptographic protocol bandwidth requirement.

Crypto. Protocol (Version/ Method)		Round Trip Cnt.	Mod. Exp. Cnt.	Pub. Key Size (bit)	Approx. Incoming Bandw. (kb/s/req)	Approx. Outgoing Bandw. (kb/s/req)
TLS	1.2	1	1	2048	3.5 – 5.5	16 – 24
				8192	3.5 – 5.5	28 – 57
	1.3		2	2048	8 – 10	15 – 27
				8192	20 – 22	21 – 34
SSH	KEX	2	2	2048	6 – 8	13 – 19
				8192	13 – 15	25 – 35
	GEX	3		2048	8 – 9	22 – 30
				8192	12 – 14	26 – 34

The bandwidth requirement can also depend on the DH key size, as the cryptographic protocol may require the client to send its public key before the server would compute its public key. When the cryptographic protocol not only refers to DH group parameters defined in standards but allows the use of custom parameters, they must be sent as part of the handshake, meaning that if both the parameters and the client's public key may be sent it requires one kilobyte extra data in every connection on both directions in the case of an 8192-bit key size.

a: EARLY VERSIONS

In early protocol versions (1.0–1.2) there is no negotiation for DH group parameters, the only DH group parameter set (p, g) is defined by the server and sent together with the server DH public key as a response to the initial client-side cryptographic handshake message (`ClientHello`) if the result of the cipher suite negotiation is one which key exchange is Diffie–Hellman. An attacker can ensure it easily if the server supports at least one DH cipher suite sending only DH cipher suites (see Section III). As a consequence of that way of working only the attacker has to send only the client hello message to trigger the public key generation.

The outbound bandwidth requirement is determined by the size of the client hello message which can be optimized as in theory it can contain only one cipher suite and no extensions meaning only 60 bytes including the TLS record header. In practice some extensions [11] are necessary (e.g. server name [12], renegotiation extension [72]) most of the time to perform a successful handshake. It means an extra 11 bytes of data plus the length of the server's fully qualified domain name for server name extension.

The inbound bandwidth requirement is determined by the server response messages as the attacker needs to wait for the message containing the server's public key (`ServerKeyExchhange`) to be sure that CPU-intensive modular exponentiation has been done by the server. Before that message server sends the initial cryptographic handshake message (`ServerHello`) which contains the selected cipher suite, which in this case must be the same as the attacker sent in client hello. The structure and also the consequence of the length of the server hello message are similar to the client hello message, especially since client hello without extensions prevents the server from sending extension in server hello.

However, the bandwidth requirement is determined by the next two messages (`ServerCertificate`, `ServerKeyExchange`). The former contains the items of the server's certificate chain in ANS.1 encoding, containing usually two (leaf, intermediate) or three (root certificate) certificates each size is at least a thousand bytes. The latter contains the DH group parameters and the server's DH public key with the same size which is usually 256 bytes (2048-bit key size), however, they can be 1024 bytes (8192-bit key size). The length of the response messages in total is at least 3-4000 bytes. Performing an extra round-trip attacker can send an arbitrary number as its public key (`ClientKeyExchange`) along with a small message signaling the transitions in ciphering strategies (`ChangeCipherSpec`) to trigger a second modular exponentiation on the server-side (shared key calculation). Attack can wait for the servers' signaling message to be sure that the second modular exponentiation has already been done.

However RFC 7919 defines negotiation of DH group parameters, it does not mean a substantial change in bandwidth requirement. The client sends the list of the supported DH groups in an extension [17] of the client's hello message. It requires a few extra bytes as the attacker uses the same method as it used in the case of the cipher suites, by sending only one DH group, the largest one the server supports. The server sends the selected DH group parameters back to the client independently from the fact that it could refer to them as the client did in its initial cryptographic handshake message to maintain compatibility with the earlier TLS versions. As a consequence, the bandwidth requirement does not change compared to the earlier TLS versions.

b: VERSION 1.3

In contrast, TLS version 1.3 allows the client to send its DH public key as an extension [71] of the client hello message, meaning that the server can calculate the shared key right after the initial handshake message received from the client. TLS 1.3 decreased the number of handshake messages. Among others, client and server key exchange messages ceased to be used, as instead of sending DH group parameters directly, they are referred identifiers, reducing the required incoming bandwidth of the attack. This way of working reduces the necessary number of round-trips to one, cutting back the latency and making the cryptographic handshake significantly effective.

At the same time, D(HE)at attack also becomes more effective, as with a single message two modular exponentiation can be triggered. The outbound bandwidth requirement is increased with the size of the DH public key compared to the one round-trip version of the attack in TLS 1.2, but the amplification factor is doubled. The incoming bandwidth requirement in the application layer is approximately the same as it was in the case of TLS 1.2, but the one round-trip reduces the TCP overhead (ACK messages).

c: OPPORTUNISTIC TLS

Several application layer protocols (e.g.: SMTP, IMAP, ...) allow the establishment of TLS channels by an extension of the original protocol, called opportunistic TLS. In this way of working the client can query the extensions supported by the server and if the TLS support is among the supported ones it can initiate to change TLS protocol, usually by sending the `STARTTLS` command. As described it usually requires two extra round-trips (query, TLS initialization) before the cryptographic handshake can start and requires a small amount – usually a few ten bytes – to be sent and received at the application level, however, extra round-trips require an extra TCP overhead (ACK messages).

d: SSH

The SSH protocol differs from TLS in several ways influencing the bandwidth requirement of a D(HE)at attack significantly. The most important difference is in the structure of the cryptographic handshake. The initial message (protocol version exchange) both on the client and server side contains only a protocol and a software version in clear text format (e.g. `SSH-2.0-OpenSSH_8.1`). The second handshake message (`KEXINIT`) contains the identifiers of the cryptographic algorithms that the peer supports. Algorithms are identified by their names instead of numeric IDs and all the supported algorithms are sent part of that message by both parties which significantly increases the required bandwidth compared to TLS. However, the attacker can send an optimized message containing only the shortest name algorithm for each type of algorithm it still requires approximately 140-150 bytes at the application level. At the same time, the attacker cannot influence the configuration of the server, meaning that it would send several algorithms in each algorithm type, requiring typically 900-1000 bytes in the application layer.

Similar to TLS 1.3 the client has to send its DH public key to the server (`KEXDH_INIT`) before it can get the server's public key, meaning that another 1024 bytes need to be sent and received by the client considering the fact that 8192-bit DH key sizes are wildly enabled on SSH servers. The server along with the DH group parameter sends its host key (host/X.509 certificates also possible, but rarely used) contains only the server's public key without any additional information (e.g. validity, extensions

in X.509, ...) or other items in the certificate chain shortening this type of message (`KEXDH_REPLY`) compared to messages (`ServerCertificate`, `ServerKeyExchange`) in TLS. As an SSH server usually has multiple host keys with different types (e.g. RSA, ECDSA, EdDSA) attacker may force the server to send the host key that has the smallest key length by sending only the necessary host key algorithm part of the key exchange initialization message. Applying these optimizations the server's key exchange reply message would contain a 1024 bytes (8192-bit) DH key, 32 host key (e.g.: ECDSA, EdDSA), and a signature, typically 32-384 bytes according to the host key type. There is another message (`NEWKEYS`) that signals that parties are ready to communicate encrypting using the new keys, meaning that the server has done the second modular exponentiation (shared key calculation).

The protocol detailed above requires higher outbound, but lower inbound bandwidth and two extra round-trips compared to the 1.3 version of TLS protocol. To reduce the number of round-trips the attacker can send the three client messages together. There is another DH key exchange method defined in the SSH protocol called group exchange, which allows the server to use custom DH group parameters. In this case an extra pair of messages (`DH_GEX_REQUEST`, `DH_GEX_GROUP`) need to be sent. The initial message contains the DH parameter sizes the client supports and the reply message contains the DH group parameters with the size selected by the server. After reconnaissance of the server configuration, a malicious client can send the largest key only enabled in the server configuration. In contrast to the latter method where an attacker can fully precalculate the client-side messages, this method requires a minimal amount of computation on the client side, as the server can send different DH group parameters – using the same size – in each connection. After receiving the DH group parameters the malicious client can send an arbitrary number less than the prime (p) in the group parameters as its public key instead of performing the modular exponentiation (see Section III). However a small enough number can be suitable for each prime number.

It should be emphasized that the bandwidth requirement in table 11 are approximate because certain values highly depend on the aforementioned cryptographic protocol details, the server configuration – such as enabled algorithms on SSH, X.509 certificate chain length in TLS – and also the lower level network parameters (e.g. MTU).

V. ATTACK SCENARIOS

In the following, the most important scenarios are described from the point of view of the victim assuming that Diffie–Hellman key exchange is enabled on a server, meaning that it is vulnerable to the D(HE)at attack.

A. THE WORST-CASE SCENARIO

The worst-case scenario arises when the protocol and its implementation offer DH parameter negotiation. This risk is even higher if the enabled parameters depend on a default setting that allows the usage of the largest modulus. The situation becomes even more critical if the cryptographic library, the application server, or its configuration is using large exponents, or if the implementation of modular exponentiation is not optimal. If DH parameter negotiation is available, a malicious client can force the largest modulus, which is mostly 8192 bits. In the worst-case scenario, the public key is paired with a private exponent using the same key size, meaning that even the most optimal library implementation can generate only just $5-12$ keys per second (see Table 7) on a single thread of a modern CPU. Using a low-end CPU the generation of a single DH public key requires $2-3$ seconds, which is dramatically slow considering the fact that these types of CPUs have only a few CPU threads.

It is crucial to highlight that a malicious client can enforce the server to both generate a public key and compute the shared key, thereby adding an extra modular exponentiation step. This action cuts the speed of the Diffie–Hellman key exchange process in half, reducing it to an average rate of $2.5-6$ exchanges per second. A modern CPU performance with 32 threads would be $32\times(2.5-6)\approx 80-190$ exchanges per second, meaning that sending at least $80-190$ TLS client hello messages per second – using approximately $1.7-6.6$ Mb per second bandwidth – would cause 100% CPU utilization. This volume of requests does not require specialized DoS capabilities. A simple Digital Ocean instance with 1 CPU and 1 GB RAM, priced at only 0.009 USD per hour (6 USD / month), may serve this purpose effectively in practice (see Section V-D). It has the capacity to overload a modern server by causing service disruptions.

It is not just a theoretical case where a large modulus is paired with a large exponent (see Table 4) and the larger exponent sizes are enabled by default (see Table 10). For instance, Ubuntu 22.04 LTS uses OpenSSL 3.0.2, which is affected by CVE-2022-40735, offers 8192-bit public key size by default and supports TLS 1.3, where a single message enough to enforce the server to generate a public key and to compute the shared secret.

B. BEST-CASE SCENARIO

To achieve a 128-bit security level, comparable to that of AES-128 for symmetric encryption or X25519/P-256 curves for asymmetric encryption, one can simply use a 3072-bit modulus in combination with a roughly 256-bit private key in the DH key exchange. There is no reason for utilizing a larger modulus in the DH key exchange when paired with AES-128, considering that AES-128 offers maximum security of 128 bits. Using an oversized modulus and private key would result in a huge dropdown in performance. By employing a cryptographic library with efficient implementation and suitable parameters, along with a well-configured application server, a significantly larger amount of requests would be necessary to achieve 100% CPU utilization.

Under the assumption that we can generate 1811 public keys per second (see Table 6, OpenSSL 3.0 column with 3072/272 public/private keys) and the CPU has 32 threads, as was the case in the previous (worst-case) scenario, it would require $(1811 \times 32)/2 = 28,976$ DH key exchanges per second to achieve maximum CPU utilization, using approximately $0.6 - 1$ Gb per second bandwidth. While this attack can still be executed from a single machine, the high number of requests can be blocked by implementing simple rate-limiting measures. For example, an IP address can be banned if it exceeds 10 requests per second. However, this limitation does not affect the ability of a distributed denial-of-service (DDoS) attack carried out by a botnet comprising approximately 3000 machines. In this case, each individual bot only needs to send 10 requests per second and $\approx 200 -$ 340 Kb per second bandwidth, imposing a negligible load on the bot itself. Consequently, the botnet can effectively target a large number of machines concurrently.

C. MOST SECURE SCENARIO

Some cryptographic algorithms – such as AES-256, Elliptic Curve Diffie–Hellman with NIST P-521 curve – can provide 256 bits of security. This security strength can be achieved in theory with the finite field Diffie–Hellman key exchange using at least 15360-bit public and at least 512-bit private keys according to NIST [4]. In practice, working with such enormous parameters is challenging because there are no standardized group parameters of that size in RFC 7919 or RFC 8268 standards. Generating group parameters of this size is possible, but it is important to note that only GnuTLS can handle Diffie–Hellman key exchange with primes of this magnitude, and this process does indeed take a substantial amount of time. Most cryptographic libraries, including OpenSSL, support a maximum prime modulus size of 10,000 bits, as shown in Table 12. Even if the libraries were compatible with a 15360-bit key size, the performance would be extremely poor. For instance, when dealing with a 15360-bit modulus and suitable 512-bit private key, the performance is roughly four times slower on a Intel Xeon E-2224 CPU @ 3.40GHz CPU compared to an 8192-bit modulus with an 8192-bit exponent (long exponent issue) as shown in Figure 12.

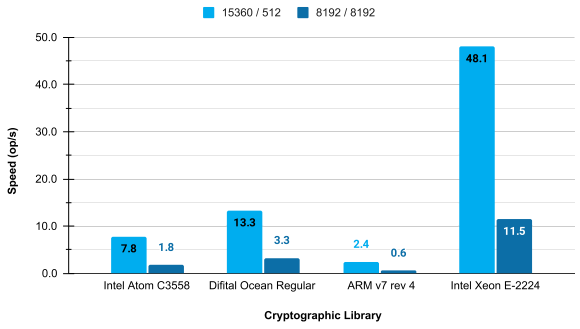

FIGURE 12. DH key generation speed with 15360/512 and 8192/8192 keys using OpenSSL 3.0 on Intel Xeon E-2224 CPU @ 3.40GHz .

TABLE 12. Cryptographic library maximum supported DHE modulus size.

Library	Max. Prime Modulus Size Supported
OpenSSL 1.0	10000 [42]
OpenSSL 1.1.1	10000 [43]
OpenSSL 3.0	10000 [44]
BoringSSL	10000 [74]
LibreSSL	10000 [75]
GnuTLS	16384 [19]
NSS	8192 [36]
Mbed TLS	8192 [79]
OpenJDK	8192 [47]
WolfSSL	≥4096 [82]

Based on our measurements and experiments, we can deduce that attaining a 256-bit security level in DH key exchange is not only a challenging endeavor but also carries a significant risk of potential DoS attacks.

D. REAL-LIFE SCENARIO

In real-world scenarios, a server's CPU utilization is typically not at zero, and the server application does several other operations than only the DH public key and shared secret calculation (see Section IV-A, making it easier for an attacker to execute their malicious actions with significantly fewer resources, as demonstrated in both the worst and best-case scenarios. We also should not forget when a 100% can be achieved the normal traffic can cause the overload. It means that during a D(HE)at attack, a malicious actor should monitor the server's response times to conclude the load on the server side and adjust the necessary number of requests as needed saving resources on the attacker's side.

In practical situations, simply banning an IP address may not always be a viable solution. This is due to the widespread use of network address translation (NAT), where hundreds or even thousands of machines share the same public IP address for internet communication. This opens up an additional attack surface for malicious users, enabling them to ban a huge number of active machines by initiating incomplete handshakes. It is worth noting that these handshakes adhere fully to established standards, making it even more challenging to detect such attacks.

VI. POSSIBLE MITIGATION SCENARIOS

Due to the high computational cost associated with modular exponentiation for large numbers, the DH key exchange is vulnerable to the modular exponential attack by design. Through minor efforts, we have successfully demonstrated in real-life scenarios that it is possible to achieve 100% CPU utilization on the latest servers, even when up-to-date patches are applied. This raises the question of what the solution might be and whether a solution exists. While the protocol is inherently vulnerable, several factors can be taken into consideration.

A. DISABLE OR RESTRICT DH KEY EXCHANGE

The most obvious mitigation technique is to disable the DH key exchange, although there is no other forward secret

alternative, just the ECDHE protocol. Moreover, entirely deactivating DH may not always be feasible due to the constraints of legacy systems. One can categorize services into two primary types: public services and private services.

1) PUBLIC SERVICES

In the case of public services, it may not always be possible to permanently deactivate DH key exchange due to legacy considerations. In public services, there is a requirement to support a large number of clients, and not all of them are prepared to use the most up-to-date protocols and cipher suites. Consequently, limiting the availability to only ECDHE protocols would render it impossible for certain clients to establish connections. Nonetheless, the risk can be mitigated by disabling DH in TLS 1.3 and restricting its usage to TLS 1.0-1.2 only. This can also have the advantage that private key sizes can be ensured by setting the optional private value length in the DH parameter file. To provide backward compatibility, `DHE` cipher suites may remain unchanged prior to TLS version 1.3, but the size of the DHE key sizes should be strictly limited by using DH parameter files with the appropriate key size. It should be noted that it is not a solution in the cases when the cryptographic library supports only negotiating the finite field DH parameters, as defined in RFC 7919.

2) PRIVATE SERVICES

In these situations, administrators typically deal with a limited number of client applications and a specific range of their versions. As a result, they can require that client applications support the latest cryptographic algorithms. In practical terms, this may involve enforcing the obligatory requirement for ECDH support, thereby enabling the complete deactivation of finite field DH key exchange algorithms on servers. Typically, it is also viable to employ the most secure algorithms without concern for their resource demands. This is achievable because both the source and frequency of resource requests can be effectively restricted, preventing the success of a DoS attack like D(HE)at.

Application servers that employ TLS for securing the confidentiality and integrity of communication channels have the capability to configure cipher suites and named groups. To deactivate finite field DH algorithms, cipher suites should either be explicitly configured to exclude DHE algorithms or implicitly configured by using predefined groups and implementing a rule to disable DH algorithms. For example, one can add the configuration line `...:!kDHE` to the server's cipher suite settings when using OpenSSL.

Named groups should also be configured explicitly while omitting algorithms such as `FFDHEx`. Application servers that implement the SSH protocol allow for the independent configuration of key exchange algorithms separate from other cryptographic algorithms. Typically, these key exchange algorithms are configured individually, so removing algorithms with names starting with `diffie-hellman` effectively disables finite field DH key exchange. Regardless of whether DH key exchange can be disabled, implementing rate limiting is always a good practice for a private service, especially when dealing with an expected lower volume of connections, which is common in the case of SSH.

B. APPROPRIATE PARAMETER SELECTION

As previously demonstrated, it is crucial to make a proper choice regarding the size of the "p" modulus in the DH key exchange. Given that the most popular cryptographic algorithms offer at least 128 bits of security, the key sizes employed in Diffie–Hellman key exchange should also guarantee a security level of at least 128 bits to prevent any reduction in security. It is essential to select the correct "p" size accordingly to avoid the unnecessary use of overly large moduli, which can result in a noticeable drop in performance. For achieving a 128-bit security level, a 3072-bit modulus is sufficient, and larger moduli can lead to a significant performance decrease. The Cryptolyzer [49] tool can, for example, be utilized to verify the supported modulus in a pre-configured server for various protocols (refer to Listing 7). This tool can assist in testing server configurations to prevent improper modulus settings and can also be employed to assess the default settings of various application servers.

```
# Installation
pip install cryptolyzer

# Check a generic TLS service (e.g. HTTPS)
cryptolyze tls1_2 dhparams example.com
cryptolyze tls1_3 dhparams example.com

# Check an opportunistic TLS service (e.g. SMTP, POP3)
cryptolyze tls1_2 dhparams \
smtp://example.com:25
cryptolyze tls1_3 dhparams \
smtp://example.com:25

# Check an OpenVPN service using UDP
cryptolyze tls1_2 dhparams \
openvpn://example.com
cryptolyze tls1_3 dhparams \
openvpn://example.com

# Check an OpenVPN service using TCP
cryptolyze tls1_2 dhparams \
openvpntcp://example.com
cryptolyze tls1_3 dhparams \
openvpntcp://example.com

# Check an SSH service
cryptolyze ssh2 dhparams example.com
```

LISTING 7. Checking involvement in D(HE)at attack using cryptoLyzer [49].

After selecting the most appropriate public key size(s) the related private key sizes should be reviewed which is not a trivial task, since the used private key sizes cannot be

determined using black box techniques. The private key must not leave the peer and from the public key, an observer cannot determine the size of the used private key, if it could it would pose a risk to the security of the key exchange. The source code of the cryptographic library and/or the application server implementation must be analyzed to determine the private key sizes in operation. In closed-source software log messages or documentation may provide information about the used private key sizes. Estimations regarding the size of the private key being used can also be derived based on the public key generation speed of a machine with a known CPU capacity.

In most cases, the size of the private key cannot be influenced without the modification of the source code of the cryptographic library and/or the application server. However, there is a rarely used exception in the case of the TLS protocol up to its 1.2 version.

In most application servers, it is possible to read the Diffie–Hellman parameters (p, g) from a designated parameter file. This file also includes an optional field, known as (`privateValueLength`), as defined in PKCS#3 [30]. If the cryptographic library is capable of recognizing this optional value, it is possible to restrict the size of the private key by setting it to an appropriate value.

The GitLab repository of D(HE)ater [50] contains parameter files with private values set to an appropriate value. These files are available for different key sizes of the well-known (RFC 7919, RFC 3526) parameters. For custom parameters or private key sizes, there is a command-line tool in D(HE)ater to check and/or set the optional private key size value of an existing DH parameter file (see Listing 8).

```
# Installation
pip install dheater

# Private key size checking
dh_param_priv_key_size_setter \
path/to/dh/parameter/file

# Private key size setting
dh_param_priv_key_size_setter \
--private-key-size {size} \
path/to/dh/parameter/file
```

LISTING 8. Setting optional private key size in a DH parameter file using D(HE)ater.

In particular, when dealing with OpenSSL versions prior to 3.0 (up to TLS 1.2), it is crucial to set this value appropriately. Failure to do so will result in the private key size being used as the public key size (see Section III-F). From version 3.0, OpenSSL uses a reduced private key size even when the (`privateValueLength`) field is absent, but this only applies if the group parameters are well known (as defined in RFC 3526, RFC 5114, or RFC 7919). It means if someone generates a DH parameter file with custom group parameters (for example using `openssl dhparam` command), the long exponent issue identified as CVE-2022-40735 is still valid. Based on our most recent evaluation of the top 1 million websites, it is estimated that approximately 8.4% utilize custom group parameters (p, g), meaning that they remain affected in long exponent attack after upgrading to OpenSSL 3.0.

C. LIMIT THE NUMBER OF CONNECTIONS

Independently from the chosen key sizes or even the key exchange algorithm, it is considered to be a good practice to rate limit the number of unauthenticated sessions an application server should handle concurrently, especially in the case of private services such as SSH. Some application servers (e.g. OpenSSH [39], [40], [41]) support that good practice in themselves, but in most cases, application servers should be integrated with an external (e.g. Fail2Ban) or an independent application (e.g. Linux Netfilter) to enforce any limitations.

Built-in mechanisms are typically the first choice for rate limiting, but custom solutions may not always be effective. In these instances, it is recommended to use third-party solutions, based on the information that originates from the application layer. Such information can be the source address of a potentially malicious client performing unsuccessful authentications, or handshakes, where the latter is a typical sign of the D(HE)at or similar DoS attacks. However, it should be noted that rate limiting and banning should be handled with due care to avoid service outages with certain customers.

VII. CONCLUSION

In this research, we conducted an in-depth exploration of a practical denial-of-service (DoS) attack against the Diffie–Hellman key exchange protocol (the D(HE)at attack). The attack allows remote users to send arbitrary numbers to the victim that are actually not public keys and trigger expensive server-side DH modular-exponentiation calculations. Our findings suggest that the D(HE)at attack is a fundamental issue related to the costly modular exponentiation. As such, devising a countermeasure is not a simple task, and needs a more profound understanding of the correct selection of group parameters.

A. LONG EXPONENT ISSUE (CVE-2022-40735)

There is a huge difference in performance in the usage of short and long exponent in the DH key exchange as demonstrated in Section IV. Certain cryptographic libraries have been found to use excessively large exponents in the DH key exchange, leading to the assignment of CVE-2022-40735. In this study, we conducted a thorough performance comparison between cryptographic libraries that utilize large exponents, deviating from NIST recommendations, and those that use appropriate private exponents.

B. 256 BIT SECURITY

Reaching 256-bit security using the finite field Diffie–Hellman key exchange is challenging. The majority

of cryptographic libraries do not support a 15360-bit modulus and there are no standardized public parameters for that size. Even when they do, the performance of the DH key exchange is dramatically low compared to the Elliptic-curve Diffie-Hellman and would hold a significant risk of a D(HE)at attack.

C. DEFAULT PARAMETERS

Cryptographic protocols can handle the negotiation of both cryptographic algorithms and their respective parameters. This means that when parameters are open to negotiation, they could include larger key parameters – such as FFDHE8192 – by default, making them more vulnerable to D(HE)at attacks. By default, WolfSSL employs a 2048-bit DH group parameter, while other libraries utilize an 8192-bit DH group parameter. This might create a misleading impression that WolfSSL is significantly faster than other libraries when in reality, it simply employs a smaller DH group parameter by default as part of its design. Figure 13 shows the DH key generation speeds that an attacker can enforce if the parameter negotiation default values are not overwritten by the application server, which is quite common.

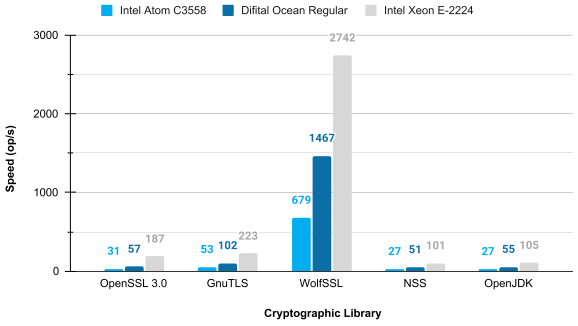

FIGURE 13. DHE key generation speeds that an attacker can enforce if negotiation parameter defaults are used on Intel Xeon E-2224 CPU @ 3.40GHz .

This example highlights again the importance of making a thoughtful choice when selecting the right cryptographic library and application server to fulfill our particular needs.

D. ELLIPTIC-CURVE DIFFIE–HELLMAN

Naturally, the question arises: Is the ECDH protocol entirely immune to such a DoS attack?

On the one hand, it is possible to select elliptic curves [7] (Brainpool curves, NIST-P curves, Bernstein curves) that provide a significantly better ratio of security strength to speed [6] compared to the finite field Diffie–Hellman key exchange. On the other hand, it should be emphasized that in the case of certain elliptic curves, the speed ratio compared to finite field DH (see Figure 14) does not seem necessarily enough to completely mitigate an elliptic-curve based D(HE)at attack, which would require further investigation not part of this study.

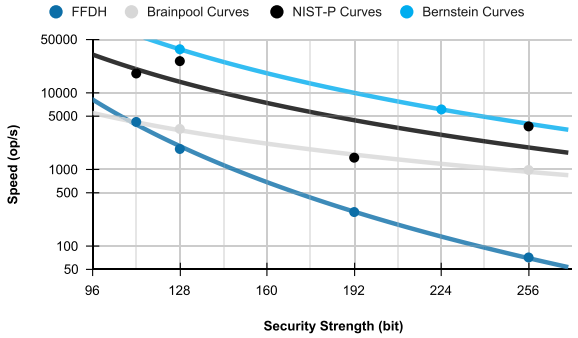

FIGURE 14. DHE key generation speeds compared to ECDHE key generation speeds on Intel Xeon E-2224 CPU @ 3.40GHz using OpenSSL.

E. TO DISABLE OR NOT TO DISABLE?

Based on this research we can conclude that finite field DH key exchange using larger key sizes is very resource-intensive which makes it vulnerable to DoS/DDoS attacks. Developers and administrators should choose both parameters and default values related to Diffie–Hellman key exchange by all three pillars of the CIA triad. Using large exponents and or offering large public key sizes by default satisfies confidentiality requirements, but poses a significant risk to the availability of their worse performance. We recommend that developers and administrators shift towards exclusively enabling algorithms that are not just effective enough to provide confidentiality, but also efficient enough to provide availability at least until cryptographic protocols do not provide effective protection against DoS/DDoS attacks in themselves. Based on this study, we are committed to conducting ongoing investigations to assess the speed and performance of different elliptic curve Diffie–Hellman key exchange protocols, and we intend to continue our research in this direction.

REFERENCES

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann, "Imperfect forward secrecy fails in practice," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Denver, CO, USA, Oct. 2015, pp. 5–17. [Online]. Available: https://dl.acm.org/doi/10.1145/2810103.2813707

[2] Cybersecurity & Infrastructure Security Agency, "Siemens SCALANCE W1750D," Tech. Rep. ICSA-22-314-10, Nov. 2022. [Online]. Available: https://www.cisa.gov/news-events/ics-advisories/icsa-22-314-10

[3] B. Barak and M. Mahmoody-Ghidary, "Merkle puzzles are optimal—An $O(n^2)$-query attack on any key exchange from a random oracle," in *Advances in Cryptology—CRYPTO 2009*, vol. 5677, S. Halevi, Ed. Berlin, Germany: Springer, 2009, pp. 374–390. [Online]. Available: http://link.springer.com/10.1007/978-3-642-03356-8_22

[4] E. Barker, "Recommendation for key management: Part 1—General," Rev. 5, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. NIST SP 800-57, Jan. 2023. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final

[5] M. Baushke, *More Modular Exponentiation (MODP) Diffie–Hellman (DH) Key Exchange (KEX) Groups for Secure Shell (SSH)*, document RFC 8268, RFC Editor, Dec. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8268.txt

[6] D. J. Bernstein, "Curve25519: New Diffie–Hellman speed records," in *Public Key Cryptography—PKC 2006*, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds. Berlin, Germany: Springer, 2006, pp. 207–228.

[7] L. Chen, D. Moody, A. Regenscheid, A. Robinson, and K. Randall, "Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters," National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. NIST SP 800-186, 2023. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-186/final

[8] T. Dierks and E. Rescorla. *Internet Key Exchange Protocol Version 2 (IKEv2)*, document RFC 5246, RFC Editor, Aug. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5246.html

[9] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, document RFC 5246, RFC Editor, Aug. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5246.html#section-7.4.7.2

[10] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. IT-22, no. 6, pp. 644–654, Nov. 1976. [Online]. Available: http://ieeexplore.ieee.org/document/1055638/

[11] D. Eastlake, *Transport Layer Security (TLS) Extensions: Extension Definitions*, document RFC 6066, RFC Editor, Jan. 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6066.html

[12] D. Eastlake, *Transport Layer Security (TLS) Extensions: Extension Definitions*, document RFC 6066, RFC Editor, Jan. 2011. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6066.html#section-11.1

[13] F5, Inc., "K83120834: Diffie–Hellman key agreement protocol weaknesses CVE-2002-20001 & CVE-2022-40735," Tech. Rep. K83120834, May 2022. [Online]. Available: https://my.f5.com/manage/s/article/K83120834

[14] M. Friedl, N. Provos, and W. Simpson, *Diffie–Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol*, document RFC 4419, RFC Editor, Mar. 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4419.txt#section-6.2

[15] M. Friedl, N. Provos, and W. Simpson, *Diffie–Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol*, document RFC 4419, RFC Editor, Mar. 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4419.txt

[16] M. Friedl, N. Provos, and W. Simpson, *Diffie–Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol*, document RFC 4419, RFC Editor, Mar. 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4419.txt#section-3.1

[17] D. Gillmor, *Negotiated Finite Field Diffie–Hellman Ephemeral Parameters for Transport Layer Security (TLS)*, document RFC 7919, RFC Editor, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7919.txt

[18] GnuTLS. *GnuTLS 3.6.0 Released*. Accessed: Dec. 28, 2023. [Online]. Available: https://lists.gnupg.org/pipermail/gnutls-devel/2017-August/008484.html

[19] GnuTLS. *Maximum Prime Modulues Size in GnuTLS Version 3.7.9*. [Online]. Available: https://gitlab.com/gnutls/gnutls/-/blob/3.7.9/lib/gnutls_int.h#L230

[20] D. Harkins and D. Carrel, *The Secure Shell (SSH) Transport Layer Protocol*, document RFC 2409, RFC Editor, Nov. 1998. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2409

[21] D. Harkins and D. Carrel, *The Secure Shell (SSH) Transport Layer Protocol*, document RFC 2409, RFC Editor, Nov. 1998. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2409#section-5.5

[22] H. Harney, U. Meth, A. Colegrove, and G. Gross, *GSAKMP: Group Secure Association Key Management Protocol*, document RFC 4535, RFC Editor, Jun. 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4535.txt

[23] Synology. (Apr. 2023). *Release Notes for Synology Mail Server*. [Online]. Available: https://www.synology.com/en-global/releaseNote/MailServer

[24] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen, *Internet Key Exchange Protocol Version 2 (IKEv2)*, document RFC 5996, RFC Editor, Sep. 2010. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5996.html

[25] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen, *Internet Key Exchange Protocol Version 2 (IKEv2)*, document RFC 5996, RFC Editor, Sep. 2010. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5996.html#section-3.3.2

[26] M. Lepinskiand and S. Kent, *Additional Diffie–Hellman Groups for Use With IETF Standards*, document RFC 5114, RFC Editor, Jan. 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5114.html

[27] E. Kiner and T. April. (Jul. 2023). *Google Mitigated the Largest DDoS Attack to Date, Peaking Above 398 Million RPS*. [Online]. Available: https://cloud.google.com/blog/products/identity-security/google-cloud-mitigated-largest-ddos-attack-peaking-above-398-million-rps

[28] T. Kivinen and M. Kojo, *More Modular Exponential (MODP) Diffie–Hellman Groups for Internet Key Exchange (IKE)*, document RFC 3526, RFC Editor, May 2003. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3526.html#section-8

[29] T. Kivinen and M. Kojo, *More Modular Exponential (MODP) Diffie–Hellman Groups for Internet Key Exchange (IKE)*, document RFC 3526, RFC Editor, May 2003. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3526

[30] RSA Laboratories. *Pkcs #3: Diffie–Hellman Keyagreement Standard*, Nov. 1993. [Online]. Available: https://www.teletrust.de/fileadmin/files/oid/oid_pkcs-3v1-4.pdf

[31] C. H. Lim and P. J. Lee, "A key recovery attack on discrete log-based schemes using a prime order subgroup," in *Advances in Cryptology—CRYPTO '97*, vol. 1294, G. Goos, J. Hartmanis, J. Van Leeuwen, and B. S. Kaliski, Eds. Berlin, Germany: Springer, 1997, pp. 249–263. [Online]. Available: http://link.springer.com/10.1007/BFb0052240

[32] Majestic-12 Ltd. *Top 1 Million Webiste*. Accessed: Dec. 28, 2023. [Online]. Available: https://majestic.com/reports/majestic-million

[33] R. C. Merkle, "Secure communications over insecure channels," *Commun. ACM*, vol. 21, no. 4, pp. 294–299, Apr. 1978. [Online]. Available: https://dl.acm.org/doi/10.1145/359460.359473

[34] Mozilla. *NSS 3.20 Release Notes—Firefox Source Docs Documentation*. Accessed: Dec. 28, 2023. [Online]. Available: https://www.devdoc.net/web/developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/NSS_3.20_release_notes.html

[35] Mozilla. *NSS 3.25 Release Notes—Firefox Source Docs Documentation*. Accessed: Dec. 28, 2023. [Online]. Available: https://www.devdoc.net/web/developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/NSS_3.25_release_notes.html

[36] Mozilla. *Maximum Public Key Size Offered by Default in NSS Version NSSVersion*. Accessed: Dec. 28, 2023. [Online]. Available: https://ithub.com/nss-dev/nss/blob/NSS_3_88_RTM/lib/ssl/sslimpl.h#L134

[37] *National Vulnerability Database (NVD)*, Standard CVE-2002-20001, National Institute of Standards and Technology, Nov. 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2002-20001

[38] *National Vulnerability Database (NVD)*, Standard CVE-2022-40735, National Institute of Standards and Technology, Nov. 2022. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-40735

[39] OpenBSD. *sshd_config(5)—OpenBSD Manual Pages—MaxStartups*. Accessed: Dec. 28, 2023. [Online]. Available: https://man7.org/linux/man-pages/man5/sshd_config.5.html#MaxStartups

[40] OpenBSD. *sshd_config(5)—OpenBSD Manual Pages—PerSourceMaxStartups*. Accessed: Dec. 28, 2023. [Online]. Available: https://man7.org/linux/man-pages/man5/sshd_config.5.html#PerSourceMaxStartups

[41] OpenBSD. *sshd_config(5)—OpenBSD Manual Pages—PerSourceNetBlockSize*. Accessed: Dec. 28, 2023. [Online]. Available: https://man7.org/linux/man-pages/man5/sshd_config.5.html#PerSourceNetBlockSize

[42] OpenSSL. *Maximum Public Key Size Offered by Default in OpenSSL Version OpenSSL Version 1.0.2*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openssl/openssl/blob/OpenSSL_1_0_2u/crypto/dh/dh.h#L77

[43] OpenSSL. *Maximum Public Key Size Offered by Default in OpenSSL Version 1.1.1t*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1s/include/openssl/dh.h#L30

[44] OpenSSL. *Maximum Public Key Size Offered by Default in OpenSSL Version 3.0.8*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openssl/openssl/blob/openssl-3.0.7/include/openssl/dh.h#L92

[45] Oracle. *JDK 11 Release Notes*. Accessed: Dec. 28, 2023. [Online]. Available: https://www.oracle.com/java/technologies/javase/11-relnote-issues.html

[46] Oracle. *JDK 9.0.4 Release Notes*. Accessed: Dec. 28, 2023. [Online]. Available: https://www.oracle.com/java/technologies/javase/9-0-4-relnotes.html

[47] Oracle. *Maximum Prime Modulues Size in OpenJDK Version 17.0.6*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openjdk/jdk/blob/jdk-19+36/src/java.base/share/classes/com/sun/crypto/provider/DHKeyPairGenerator.java#L77

[48] H. Orman and P. Hoffman, *Determining Strengths for Public Keys Used for Exchanging Symmetric Keys*, document RFC 3766, RFC Editor, Apr. 2004. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3766#section-3.1

[49] S. Pfeiffer. *CryptoLyzer: Fast and Flexible Cryptographic Settings Analyzer Library for Python With CLI*. Accessed: Dec. 28, 2023. [Online]. Available: https://cryptolyzer.readthedocs.io/

[50] S. Pfeiffer. *D(HE)ater*. Accessed: Dec. 28, 2023. [Online]. Available: https://dheatattack.gitlab.io/dheater/

[51] S. Pohlig and M. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (corresp.)," *IEEE Trans. Inf. Theory*, vol. 24, no. 1, pp. 106–110, Jan. 1978. [Online]. Available: http://ieeexplore.ieee.org/document/1055817/
[52] GnuTLS Project. *Diffie–Hellman Private Key Size Calculation in GnuTLS Version 3.7.9*. [Online]. Available: https://gitlab.com/gnutls/gnutls/-/blob/3.7.9/lib/algorithms/groups.c#L135-186
[53] GnuTLS Project. *Maximum Public Key Size Offered by Default in GnuTLS 3.7.9*. Accessed: Dec. 28, 2023. [Online]. Available: https://gitlab.com/gnutls/gnutls/-/blob/3.7.9/lib/priority.c#L161-178
[54] MbedTLS Project. *Diffie–Hellman Private Key Size Calculation in MbedTLSName Version 3.4.0*. [Online]. Available: https://github.com/Mbed-TLS/mbedtls/blob/v3.4.0/library/dhm.c#L122
[55] NSS Project. *Diffie–Hellman Private Key Size Calculation in NSS Version 3.13*. Accessed: Dec. 28, 2023. [Online]. Available: https://hg.mozilla.org/projects/nss/file/NSS_3_13_RTM/security/nss/lib/freebl/dh.c#l56
[56] NSS Project. *Diffie–Hellman Private Key Size Calculation in NSS Version 3.14*. Accessed: Dec. 28, 2023. [Online]. Available: https://hg.mozilla.org/projects/nss/file/NSS_3_14_RTM/security/nss/lib/freebl/dh.c#l28
[57] NSS Project. *Maximum Public Key Size Offered by Default in NSS Version 3.14*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/nss-dev/nss/blob/NSS_3_88_RTM/lib/ssl/sslsock.c#L2090-L2093
[58] OpenJDK Project. *Diffie–Hellman Private Key Size Calculation in OpenJDKName Version 17.0.5*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openjdk/jdk/blob/jdk-21+5/src/java.base/share/classes/com/sun/crypto/provider/DHKeyPairGenerator.java#L175
[59] OpenJDK Project. *Diffie–Hellman Private Key Size Calculation in OpenJDKName Version 17.0.6*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openjdk/jdk/blob/jdk-21+6/src/java.base/share/classes/sun/security/util/SecurityProviderConstants.java#L106-L140
[60] OpenJDK Project. *Maximum Public Key Size Offered by Default in OpenJDKName Version 17.0.6*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openjdk/jdk/blob/jdk-19+36/src/java.base/share/classes/sun/security/ssl/SupportedGroupsExtension.java#L204-L224
[61] OpenSSL Project. *OpenSSL Changes*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openssl/openssl/blob/master/CHANGES.md#user-content-changes-between-111-and-300-7-sep-2021
[62] OpenSSL Project. *Diffie–Hellman Private Key Size Calculation in OpenSSL Version 1.1.1t*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openssl/openssl/blob/OpenSSL_1_1_1q/crypto/dh/dh_rfc7919.c#L28
[63] OpenSSL Project. *Diffie–Hellman Private Key Size Calculation in OpenSSL Version 3.0.6*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openssl/openssl/blob/openssl-3.0.6/crypto/ffc/ffc_dh.c#L60
[64] OpenSSL Project. *Diffie–Hellman Private Key Size Calculation in OpenSSL Versions 3.0.5*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openssl/openssl/blob/openssl-3.0.5/crypto/ffc/ffc_dh.c#L56
[65] OpenSSL Project. *Maximum Public Key Size Offered by Default in OpenSSL Version 3.0.8*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/openssl/openssl/blob/openssl-3.0.7/ssl/t1_lib.c#L195-L213
[66] WolfSSL Project. *Diffie–Hellman Private Key Size Calculation in WolfSSL Version 5.6.0*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/wolfSSL/wolfssl/blob/v5.6.0-stable/wolfcrypt/src/dh.c#L1240-L1264
[67] WolfSSL Project. *Maximum Public Key Size Offered by Default in WolfSSL Version 5.6.0*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/wolfSSL/wolfssl/blob/v5.6.0-stable/configure.ac#L5066-L5073
[68] J.-F. Raymond and A. Stiglic, "Security issues in the Diffie–Hellman key agreement protocol," *IEEE Trans. Inf. Theory*, vol. 22, pp. 1–17, 2002. [Online]. Available: https://www.researchgate.net/publication/2401745_Security_Issues_in_the_Diffie-Hellman_Key_Agreement_Protocol
[69] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, document RFC 8446, RFC Editor, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446.txt
[70] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, document RFC 8446, RFC Editor, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446#section-4.2.7
[71] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, document RFC 8446, RFC Editor, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446#section-4.2.8
[72] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov, *Transport Layer Security (TLS) Renegotiation Indication Extension*, document RFC 5746, RFC Editor, Feb. 2010. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5746.html
[73] Shodan.io. *Shodan Search Engine*. Accessed: Dec. 28, 2023. [Online]. Available: https://www.shodan.io/search?query=ssh+diffie-hellman
[74] Boring SSL. *Maximum Prime Modulues Size in BoringSSL Version 3.8*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/google/boringssl/blob/refs/tags/fips-20220613/crypto/fipsmodule/dh/dh.c#L73
[75] Libre SSL. *Maximum Prime Modulues Size in LibreSSL Version 3.7.2*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/libressl/openbsd/blob/libressl-v3.7.2/src/lib/libcrypto/dh/dh.h#L77
[76] The PARI Group, Univ. Bordeaux. (2022). *PARI/GPversion 2.15.2*. [Online]. Available: http://pari.math.u-bordeaux.fr/
[77] M. Thomson and S. Turner, *Using TLS to Secure QUIC*, document RFC 9001, RFC Editor, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9001
[78] Mbed TLS. *Mbed TLS V3.2.1 Source Code Documentation*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/Mbed-TLS/mbedtls/blob/v3.2.1/docs/architecture/tls13-support.md
[79] Mbed TLS. *Maximum Public Key Size Offered by Default in Mbed TLS Version 3.4.0*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/Mbed-TLS/mbedtls/blob/v3.4.0/include/mbedtls/bignum.h#L85-L88
[80] P. C. Van Oorschot and M. J. Wiener, "On Diffie–Hellman key agreement with short exponents," in *Advances in Cryptology—EUROCRYPT '96*, vol. 1070, G. Goos, J. Hartmanis, J. Van Leeuwen, and U. Maurer, Eds. Berlin, Germany: Springer, 1996, pp. 332–343. [Online]. Available: http://link.springer.com/10.1007/3-540-68339-9_29
[81] wolfSSL. *WolfSSL Changelog*. Accessed: Dec. 28, 2023. [Online]. Available: https://www.wolfssl.com/docs/wolfssl-changelog/
[82] wolfSSL. *Maximum Public Key Size Offered by Default in WolfSSL Version 5.6.0*. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/wolfSSL/wolfssl/blob/v5.6.0-stable/wolfssl/internal.h#L1137-L1154
[83] T. Ylonen Ed. and C. Lonvick, *The Secure Shell (SSH) Transport Layer Protocol*, document RFC 4253, RFC Editor, Jan. 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4253.txt
[84] T. Ylonen Ed. and C. Lonvick, *The Secure Shell (SSH) Transport Layer Protocol*, document RFC 4253, RFC Editor, Jan. 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4253#section-7.1
[85] T. Ylonen Ed. and C. Lonvick, *The Secure Shell (SSH) Transport Layer Protocol*, document RFC 4253, RFC Editor, Jan. 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4253#section-8
[86] O. Yoachimik and J. Pacheco. (Oct. 2023). *DDoS Threat Report for 2023 Q2*. [Online]. Available: https://blog.cloudflare.com/ddos-threat-report-2023-q2/

SZILÁRD PFEIFFER received the B.Sc. degree in electrical engineering from Kandó Kálmán Faculty of Electrical Engineering, Óbuda University, Budapest, Hungary, in 2003. Currently, he is the Security Researcher and an Evangelist with Balasys, Budapest. His research interests include public key infrastructure, security extensions for network protocols, cryptographic protocols, and their implementations.

NORBERT TIHANYI (Member, IEEE) received the B.Sc. degree in security engineering, the M.Sc. degree in safety engineering, the M.Sc. degree in IT engineering (Hons.), and the Ph.D. degree in information science and technology from Eötvös Loránd University, Budapest, Hungary, in 2020. Currently, he is a Lead Researcher with the Technology Innovation Institute (TII), Abu Dhabi, United Arab Emirates. His research interests include cryptoanalysis, the security of embedded devices, web applications, and cryptography-related prime number theory. He is also a Public Body Member of the Hungarian Academy of Science.