RSA, DH and DSA in the Wild*

Nadia Heninger

University of California, San Diego, USA

1 Introduction

The previous chapters discussed breaking practical cryptographic systems by solving the mathematical problems directly. This chapter outlines techniques for breaking cryptography by taking advantage of implementation mistakes made in practice, with a focus on those that exploit the mathematical structure of the most widely used public-key primitives.

In this chapter, we will mostly focus on public-key cryptography as it is used on the Internet, because as attack researchers we have the greatest visibility into Internet-connected hosts speaking cryptographic protocols.

While this chapter was being written, prior to the standardisation or wide-scale deployment of post-quantum cryptography, the set of algorithms used for public-key cryptography on the Internet was surprisingly limited.

In the context of communications protocols such as the transport layer security (TLS), secure shell (SSH), and Internet protocol security (IPsec), key exchange is accomplished by using finite-field Diffie–Hellman (DH), elliptic-curve Diffie–Hellman (ECDH), or Rivest–Shamir–Adleman (RSA) encryption. Historically, RSA encryption was the most popular key-exchange method used for TLS, while SSH and IPsec only supported Diffie–Hellman. It is only in the past few years that ECDH key exchange has begun to be more popular for all three protocols. The digital signature algorithms most widely used in practice are RSA, elliptic-curve DSA (ECDSA), and prime field DSA. RSA is by far the most popular digital signature algorithm in use, and is only just now beginning to be supplanted by ECDSA for network protocols.

2 RSA

Historically, RSA has been by far the most common public-key encryption method, as well as the most popular digital signature scheme, and it remains extremely commonly used in the wild. Ninety percent of the HTTPS certificates seen by the ICSI Certificate Notary use RSA for digital signatures [ICS20]. Both TLS 1.1 [DR06] (standardised in 2006) and 1.2 [DR08] (standardised in 2008) require TLS-compliant implementations to support RSA for key-exchange. RSA encryption and signatures are also still widely used for PGP. Both host and client authentication via RSA digital signatures were recommended in the original SSH specifications [YL06], and remain in common use today [Mil20]. RSA also remains ubiquitous for other authentication scenarios, including smart cards and code signing.

*This material has been published in revised form in Computational Cryptography edited by Joppe W. Bos and Martijn Stam and published by Cambridge University Press. See www.cambridge.org/9781108795937.

2.1 RSA Key Generation

RSA Key Generation, in Theory Textbook RSA key generation works as follows: first, one generates two primes p and q of equal size, and verifies that $p-1$ and $q-1$ are relatively prime to the desired public exponent e. Then one computes the modulus $N = pq$ and private exponent $d = e^{-1} \bmod (p-1)(q-1)$. The public key is then the pair (e, N) and the private key is the pair (d, N) [RSA78].

Factorisation of the modulus N remains the most straightforward method of attack against RSA, although this is not known to be equivalent to breaking RSA [BV98, AM09].

RSA Key Generation, in Practice Both textbook RSA and the description of RSA in many standards leave a number of choices to implementers. In addition, there are several unusual properties of most RSA implementations that are surprising from a purely theoretical point of view.

2.2 Prime Generation

The simplest method for an implementation to generate a random prime is to seed a pseudorandom-number generator (PRNG), read a bit string out of the PRNG of the desired length of the prime, interpret the bit string as an integer, and then test the integer for primality by using, typically, a combination of trial division and some number of iterations of a probabilistic primality test such as Miller–Rabin [Mil76, Rab80]. If the number is determined to be composite, a new sequence of bits is read from the pseudorandom-number generator and the primality tests are repeated until a prime is found. Alternatively, an implementation may start at a pseudorandomly generated integer and increment or sieve through successive integers within some range until a prime is found. The prime-generation process varies across implementations in different cryptographic libraries [AMPS18].

2.3 Prime Structure

Many implementations enforce additional structural properties of the primes that they generate for RSA. Common properties include being 'safe' primes, that is, that $(p-1)/2$ is also a prime, or that $(p-1)$ and/or $(p+1)$ have known prime factors of some minimum size. For example, the United States National Institute of Standards and Technology (NIST) recommends that these 'auxiliary' prime factors should be at least 140 bits for the 1024-bit primes used to construct a 2048-bit RSA key [Inf13]. This is intended to protect against Pollard's $p-1$ and $p+1$ factoring algorithms [Pol74]. For the 1024-bit and larger key sizes common in practice, random primes are unlikely to have the required structure to admit an efficient attack via these algorithms, so enforcing a particular form for the primes used for RSA is less important.

Implementation Fingerprints The different choices that implementations make in generating primes can result in quite different distributions of properties among the primes generated by these distinct implementations. Some of these are detectable only from the prime factorisation, others are evident from the public key.

Mironov [Mir12] observed that OpenSSL (Open Secure Sockets Layer), by default, generates primes p that have the property that $p \neq 1 \bmod 3, p \neq 1 \bmod 5, \ldots, p \neq 1 \bmod 17\,863$; that is, that $p-1$ is not divisible by any of the first 2048 primes p_i. The probability that a random 512-bit prime has this property is $\prod_{i=2}^{2048}(p_i-2)/(p_i-1) \approx 7.5\%$, so the probability that a random 1024-bit RSA modulus has this property is 0.05625%, and thus a factored RSA key can be identified as having been generated by OpenSSL or not with good probability.

Svenda *et al.* [SNS+16] documented numerous implementation artifacts that could be used to identify the implementation used to generate a key or a collection of keys. Many libraries generated primes with distinctive distributions of prime factors of $p-1$ and $p+1$, either eliminating small primes, enforcing large prime divisors of a given size, or some combination of the two; some libraries clamped the most significant bits of the primes p and q to fixed values such as 1111 and 1001, resulting in a predictable distribution of most significant bits of the public modulus.

The ability to fingerprint implementations does not seem to be an immediate vulnerability for RSA encryption or signatures, although for some applications of RSA, where, for example, a user might use a public key as a pseudonymous identifier, the loss of privacy due to implementation fingerprinting may constitute a vulnerability. However, this analysis later led to the discovery of ROCA ('Return of Coppersmith's Attack'), which exploited the fact that the specific prime-generation process used by Infineon smart cards resulted in so much structure to the prime that the resulting RSA keys were efficiently factorable using a variant of Coppersmith's lattice methods [NSS+17].

2.4 Prime Sizes

RSA moduli are almost universally generated from two equal-sized primes. However, exceptions occasionally arise. There are a small number of RSA moduli that have been found to be divisible by very small primes: Lenstra *et al.* [LHA+12] report finding 171 RSA moduli used for HTTPS and PGP with prime factors less than 2^{24} in 2012, and Heninger *et al.* [HDWH12] report finding 12 SSH host keys that were divisible by very small primes. Many of these may be due to copy–paste or memory errors in an otherwise well-formed modulus; several of the moduli with small factors were one hex character different from another valid modulus they observed in use, or contained unlikely sequences of bytes.

Anecdotally, at least one RSA implementation accidentally generated prime factors of unequal sizes, because the library had accidentally fixed the size of one prime to 256 bits, and then generated an n-bit RSA modulus by generating a second prime of length $n-256$ [Koc20].

2.5 Short Modulus Lengths

In 1999, 512-bit RSA was first factored [CDL+00] and, by 2015, 512-bit factorisation was achievable within a few hours using relatively modest computing resources [VCL+16]. The use of 1024-bit RSA moduli was allowed by NIST recommendations until 2010, deprecated until 2013, and disallowed after 2013 [BR11]. In 2009, 1024-bit RSA was already believed to be feasible in principle to a powerful adversary using only general-purpose computing, although such a calculation was thought to be infeasible for an academic community effort for the near future, and as of this writing no factorisation of a generic 1024-bit RSA modulus has been reported yet in the public literature [BKK+09].

However, 512- and 1024-bit RSA keys remained in common use well after these dates for a number of reasons including hard-coded limits on key sizes and long-term keys that were difficult to deprecate. Multiple long-term TLS certificate authority root certificates with 1024-bit RSA keys and multi-decade validity periods remained trusted by major browsers as late as 2015 [Wil14], and the process of certificate authorities phasing out end-user certificates with 1024-bit RSA keys was still in process years afterward. DNSSEC keys are practically limited to 1024-bit or shorter RSA signing keys because many older router and network middlebox implementations cannot handle User Datagram Protocol (UDP) packets larger than 1500 bytes [BD11]. Because of these limitations, and the fact that DNSSEC predates widespread support for ECDSA, which has much shorter keys and signatures, tens of thousands of 512-bit RSA public keys were still in use for DNSSEC in

2015 [VCL+16]. A few thousand 512-bit RSA keys were still in use in 2015 for both HTTPS and mail protocols, and millions of HTTPS and mail servers still supported insecure 512-bit 'export'-grade RSA cipher suites [DAM+15, APPVP15, VCL+16]. As of 2020, support for 512-bit RSA has dropped to less than 1% of popular HTTPS sites [Qua20].

There are multiple structural causes contributing to the long lifespans of short keys. The choice of public-key length is typically left to the party generating the key. Since RSA keys were historically considered to be computationally expensive to generate, RSA key pairs tend to be infrequently generated and valid for long periods. In order to maintain interoperability, most implementations have traditionally been permissive in the lengths of RSA moduli they will accept.

Pathologically short keys are also occasionally found in practice, presumably as a result of implementers who do not understand the security requirements for factorisation-based cryptosystems. A 128-bit RSA modulus used for the DKIM protocol used to authenticate email senders was in use in 2015 [VCL+16]. In 2016, an implantable cardiac device made by St. Jude Medical was found to be secured using 24-bit RSA [Liv16]. (The US FDA later issued a recall.)

2.6 Public Exponent Choice

In theory, the public exponent e could have any length, and RSA is not known to be insecure in general with most possible choices of e, either large or small. In practice, however, implementations nearly universally use short exponents, and in fact typically restrict themselves to a handful of extremely common values.

Common Exponents The public exponent e does not contain any secret information, and does not need to be unique across keys. By far the most commonly used RSA public exponent is the value $65\,537 = 2^{16} + 1$, which has the virtue of being relatively small and has low Hamming weight, so that encryption and signature verification are both fast.

Short Public Exponents Although very small e such as $e = 3$ are not known to be broken in general, the existence of several low-exponent RSA attacks such as Coppersmith's low-exponent decryption attacks [Cop97] and Bleichenbacher's low-exponent signature forgery attacks [Fin06] makes many practitioners nervous about such values. For example, NIST requires $e > 2^{16}$ [BCR+19], which is believed to be large enough to render even hypothesised improvements to these attacks infeasible.

There are multiple barriers to using larger RSA public exponents in practice. The Windows CryptoAPI used in Internet Explorer until as late as Windows 7 encodes RSA public exponents into a single 32-bit word, and cannot process a public key with a larger exponent. NIST requires that RSA public exponents be at most 256 bits [BCR+19].

The Case $e = 1$ Occasionally implementers do choose terrible values for e: in 2013, the Python SaltStack project fixed a bug that had set $e = 1$ [tb13], and Lenstra *et al.* [LHA+12] found eight PGP RSA keys using exponent $e = 1$ in 2012. In this case, an encrypted 'ciphertext' would simply be the padded message itself, thus visible to any attacker, and a 'signature' would be the padded hash of the message, trivially forgeable to any attacker.

2.7 Repeated RSA Moduli

If two parties share the same public RSA modulus N, then both parties know the corresponding private keys, and thus may decrypt each others' traffic and forge digital signatures for each other. Thus, in theory, one would expect RSA public moduli to be

unique in the wild. In practice, however, it turns out that repeated RSA public keys are quite common across the Internet.

In 2012, Lenstra *et al.* [LHA+12] found that 4% of the distinct certificates used for HTTPS shared an RSA modulus N with another certificate. They also found a very small number (28 out of 400 000) RSA public keys that were shared among PGP users. Heninger *et al.* [HDWH12] performed a similar independent analysis in 2012, and found a rate of 5% of public key sharing in distinct HTTPS certificates. Among 12.8 million distinct hosts on the Internet who successfully completed a HTTPS TLS handshake in a single scan, there were 5.9 million distinct certificates, of which 5.6 million contained distinct public keys. Similarly, for the 10.2 million hosts who successfully completed an SSH handshake, there were only 6.6 million distinct RSA public host keys. In other words, 60% of HTTPS IPv4 hosts and 65% of SSH IPv4 hosts shared their RSA public keys with some other host on the Internet.

There are numerous reasons why keys are shared across distinct certificates, certificates are shared across seemingly unrelated hosts, and host keys are shared across thousands of hosts, not all of which are vulnerabilities. Many large hosting providers use the same backend infrastructure for large ranges of IP addresses or seemingly distinct websites. However, there are also many common reasons for shared public keys that do constitute vulnerabilities. Many home routers, firewalls, and 'Internet of things' devices come with pre-configured manufacturer default public and private keys and certificates, which may be shared across the entire device or product line for that manufacturer. These private keys are thus accessible to anyone who extracts them from one of these devices. Knowledge of the private key would allow an attacker to decrypt ciphertexts encrypted to the public key, or forge signatures that will validate with the public key. Databases of such keys have been published on the Internet [Hef10].

Random Number Generation Vulnerabilities Different entities may share identical public moduli because of random-number generation (RNG) vulnerabilities. If two different entities seed the same PRNG algorithm with the same value, then they will each obtain the same sequence of output bits from the algorithm. If two entities use the same PRNG seeds in the course of generating primes for RSA key generation, then they will both obtain the same resulting primes as output, and thus generate the same public modulus.

In 2008, Luciano Bello discovered that the Debian version of the OpenSSL library had accidentally removed all sources of entropy for the random-number generator except for the process ID [YRS+09]. This meant that only 16 384 possible RSA moduli could be generated for a given CPU architecture (32-bit or 64-bit, and big or little endian). Reportedly, he discovered the vulnerability after observing the same public keys being generated in the wild. This bug affected all cryptographic key generation between 2006 and 2008 on affected systems, and vulnerable keys were still being found in the wild years after the bug was fixed [DWH13].

The analyses of Lenstra *et al.* [LHA+12] and Heninger *et al.* [HDWH12] showed that repeated RSA public keys due to random-number generation vulnerabilities occur surprisingly often in practice. Heninger *et al.* [HDWH12] traced many of the vulnerable keys back to a flaw in the Linux random-number generator that affected many network devices. First, cryptographic keys for network services like HTTPS and SSH were often generated the first time a machine boots. Second, the Linux operating system PRNG enforced delays between seeding intervals in order to prevent attacks where a local attacker could brute force individual inputs and thus predict future outputs. Third, network devices lacked the entropy sources such as keyboard timings, mouse movements, or hard-disk timings that the operating system was using to seed the random-number generator. This resulted in a vulnerability where small devices may not yet have seeded the PRNG with any inputs from the environment when the key generation process was run.

In principle, an attacker with the ability to study a given implementation using such poor randomness could reverse-engineer the possible seeds used to generate the key pairs, and thus compute the private key corresponding to a vulnerable public key. This has been done for vulnerable Debian OpenSSL keys [YRS+09], but we are not aware of attempts to do this for implementations affected by the Linux boot-time kernel vulnerability.

The presence of a vulnerable public key is also a signal to an attacker that other non-public values that were generated by the same random-number generator are likely to be vulnerable to enumeration as well. The collisions between public keys signal that the random-number generator design or implementation is not incorporating enough entropy to have forward or backward secrecy. This means that an attacker may be able to use a public value to verify that they have successfully recovered the state of the random-number generator, and then wind the state of the random-number generator forward or backward to recover other sensitive values. This is similar in spirit to attacks on cryptographic protocols targeting flawed random-number generation designs that exploit nonces, cookies, or other public values to recover secret values generated later [CNE+14, CMG+16, CGH18, CKP+20].

The Linux kernel vulnerability was patched in 2012, and in 2014 Linux introduced a new system call with a safer interface for generating pseudorandom numbers [Edg14].

2.8 RSA Moduli with Common Factors

A more serious version of the repeated-key vulnerability arises if two different parties have different public RSA moduli N_1 and N_2 that share exactly one prime factor p in common, but have distinct second prime factors q_1 and q_2. In that case, any external attacker can compute the private keys for each party outright by computing $p = \gcd(N_1, N_2)$.

Lenstra *et al.* [LHA+12] and Heninger *et al.* [HDWH12] both independently searched for the existence of such keys in PGP, HTTPS, and SSH RSA public keys in 2012, and found that two PGP users ([LHA+12]), 0.2% of HTTPS certificates ([LHA+12]) or 0.5% of HTTPS hosts ([HDWH12]), and 0.03% of SSH hosts ([HDWH12]) had RSA public keys that were completely factored by sharing one prime factor with another RSA public key in their datasets.

This vulnerability was traced in many cases to a variant of the PRNG implementation problems above [HDWH12]. Most practical PRNG implementations mix new entropy inputs into their state during the course of normal operation. For example, the OpenSSL PRNG mixes the current time in seconds into the state of its PRNG after every call to extract an output to generate a bignum integer. If two different entities begin the RSA key-generation process using the same initial PRNG seed values, but sometime during key generation mix in different entropy inputs, the stream of PRNG outputs, and therefore the values of the primes generated, will diverge from that point onward. If the PRNG states are identical during the generation of the first prime p but diverge during the generation of the second prime q, then this results in exactly the GCD vulnerability.

A 2016 followup study by Hastings *et al.* [HFH16] gives details on the rates of vulnerable keys over time, broken down by product vendor. They observed no evidence of end users patching devices to remove vulnerable keys after vendors released vulnerability disclosures and patches, and found multiple implementations that had newly introduced GCD vulnerabilities since 2012, most likely the result of using old versions of the Linux kernel. They were able to fingerprint 95% of the vulnerable HTTPS certificates as having been generated by OpenSSL using the OpenSSL prime fingerprint discussed previously in Section 2.3. Thus the popularity of OpenSSL, together with its vulnerable pattern of behaviour, appears to have contributed to the different vulnerability rates between keys used for HTTPS and SSH.

Anecdotally, this vulnerability has also arisen in an implementation that periodically generated new RSA keys in an idle process in the background, but sometimes only one

new prime was swapped out of memory before the key was exported for use [Koc20].

2.9 RSA Primes with Shared or Predictable Bits

Heninger *et al.* [HDWH12] and Bernstein *et al.* [BCC+13] document a further variant of these shared-key vulnerabilities: they observed RSA keys in the wild whose prime factors shared most significant bits in common. If enough bits are shared in common, these RSA keys can be efficiently factored using lattice basis reduction, using techniques from Coppersmith [Cop97] or Howgrave-Graham [HG01].

For the vulnerable HTTPS and SSH RSA keys of this form documented by Heninger *et al.* [HDWH12], these primes may be due to an implementation that uses a PRNG whose output length is less than the length of the prime factor to be generated, so that multiple PRNG outputs are concatenated to generate a single prime, and the states of two different entities' PRNGs diverged during the generation of this prime. Bernstein *et al.* [BCC+13] found such primes generated by smart cards, where the prime factors appeared to be generated by a faulty physical random-number generator process that would sometimes generate predictable repeating sequences of bits in the resulting primes.

Heninger *et al.* [HDWH12] report several SSH public host keys with prime factors that were all zeros except that the first two bytes and last three bytes were set. This may have resulted from an implementation that generated primes by setting the most significant bits of a buffer as many popular implementations do [SNS+16], reading intermediate bits from a broken random-number generator that returned all zeros, and then incrementing until a prime was found. Keys of this form would be vulnerable to factoring via brute-force enumeration as well as variants of lattice attacks exploiting known bits [HG01, BCC+13].

2.10 RSA Encryption and Signing

Encryption and Signing, in Theory In theory, one encrypts a message m by calculating the ciphertext $c = m^e \bmod N$. The plaintext can be recovered by computing $m = c^d \bmod N$. For digital signatures, the signature is $s = m^d \bmod N$ and one can verify the signature by verifying that $m = s^e \bmod N$.

Encryption and Signing, in Practice In practice, RSA encryption and digital signatures must use a padding scheme to avoid a wide variety of malleability and signature forgery attacks. The PKCS#1v1.5 padding scheme [Kal98] remains almost universally in use for both encryption and digital signature padding despite the fact that PKCS#1v1.5 is not CCA secure and later versions of the PKCS#1 standard [MKJR16] included padding schemes for RSA encryption and signatures that were designed to be provably secure (OAEP [BR95] and PSS [Bel98]). Although the publication of practical padding oracle attacks against PKCS#1v1.5 [Ble98] and the development of the provably secure OAEP padding scheme both pre-dated the standardisation of TLS 1.0 in 1999, TLS 1.0–1.2 continued to use PKCS#1v1.5 in order to preserve backwards compatibility, and attempted to mitigate the threat of padding oracles with protocol-level countermeasures [DA99].

Hybrid Encryption RSA public-key encryption is almost universally used for key encapsulation for hybrid encryption in practice, where the public-key encryption operation is used only to transmit symmetric key material, and the actual encrypted content is encrypted using a symmetric encryption scheme. One prominent counterexample is the original Apple iMessage protocol, which encrypted the AES session key as well as the first 101 bytes of the symmetrically encrypted ciphertext in the RSA-OAEP-encrypted message. Unfortunately, the protocol was insecure against chosen ciphertext attacks because it did not authenticate the symmetric encryption properly [GGK+16].

2.11 Key Re-Use across Protocols

Cryptographic best practice dictates that cryptographic keys should be used for a single purpose, but this principle is not always followed. In TLS versions 1.2 [DR08] and below, a server typically has one certificate containing its public key, and this key is used both for digital signatures and encryption. The server generates digital signatures to sign Diffie–Hellman key-exchange parameters when Diffie–Hellman key exchange is used, and uses the same RSA key to decrypt secrets when RSA is used as a key exchange mechanism.

RSA keys are also universally re-used by web servers across different versions of SSL/TLS, and it is quite common for web servers to support many old protocol versions for backwards compatibility. This led to the DROWN vulnerability [ASS+16] discussed in Section 2.13, where an attacker could exploit a protocol flaw in SSLv2 to compromise an otherwise secure RSA key-exchange message from a server using the same key with TLS 1.2.

There are also sporadic cases of keys shared across entirely different protocols: Heninger *et al.* [HDWH12] document two RSA public keys that were used for both HTTPS and SSH in 2012.

2.12 Encryption Padding

In PKCS#1v1.5 [Kal98], data D is padded before encryption as

```
EB = 00 || 02 || PS || 00 || D
```

where `00` and `02` represent byte strings, `PS` is a pseudorandomly generated padding string that contains no null bytes, and the `D` is the data to be encrypted, typically a value like a TLS premaster secret from which symmetric encryption and authentication keys will be derived. Textbook RSA encryption is applied to the padded value `EB`. For decryption, textbook RSA decryption is applied to recover the padded plaintext, and the decrypter must then check that the padding is valid before stripping off the padding and returning the unpadded data.

2.13 RSA Encryption Padding Oracle Attacks

If a decryption implementation returns an error when the plaintext padding is incorrect, then it may be exploitable as an oracle for a chosen ciphertext attack. Bleichenbacher developed a padding oracle attack in 1998 against the PKCS#1v1.5 encryption padding scheme as used in SSL [Ble98]. Bleichenbacher's attack exploits implementations where the decrypter (usually a server receiving ciphertexts from a client) provides an error message if decryption fails due to incorrect plaintext padding. The attacker begins with a target ciphertext c that they wish to decrypt to recover padded message m, and submits mauled ciphertexts $a^e c \bmod N$ to the decryption oracle for carefully chosen values of a. The error messages from the padding oracle reveal whether the most significant bytes of the mauled plaintext am are `00 || 02`, which allows the attacker to iteratively narrow down possible values for m. The originally published attack required millions of queries to recover a plaintext.

Despite the existence of this attack, the designers of TLS 1.0 through 1.2 decided against using a CCA-secure RSA padding scheme, and instead continued to specify PKCS#1v1.5 padding for RSA encryption for backwards compatibility reasons. To attempt to mitigate padding oracle attacks, the TLS standards required that when implementations encountered RSA padding errors on decryption, they should avoid signaling the error to the attacker by simply generating a placeholder plaintext and continuing with the handshake using this value. In this case, the handshake should naturally fail when the client sends its authentication of the handshake. However, numerous studies have found

that implementation support for this countermeasure is often incomplete, and many implementations of this countermeasure result in side channels that lead to practically exploitable padding oracle attacks in the wild [MSW+14, BSY18].

The 2015 DROWN attack [ASS+16] exploited a confluence of vulnerabilities in RSA usage in the wild: (1) RSA keys were universally used by servers across different versions of the TLS protocol including old versions of SSL, (2) the SSLv2 protocol supported weakened key strengths at the protocol level in order to conform to the US government regulations on the export of cryptography, (3) many servers never disabled support for SSLv2 for backwards compatibility reasons and (4) both a protocol flaw in SSLv2 and implementation flaws in the OpenSSL library could serve as particularly strong padding oracles that could allow an attacker to decrypt TLS 1.2 ciphertexts or forge digital signatures after tens of thousands of connections. In 2015, 33% of HTTPS servers were vulnerable to this attack [ASS+16].

OAEP is also vulnerable to padding oracle attacks if implemented improperly. In 2001, Manger gave a chosen ciphertext attack against OAEP that can recover plaintext after as few as 1000 queries if the implementation allows an attacker to distinguish between integer-byte conversion errors and integrity check errors, a situation that was not ruled out by PKCS#1v2.0 [Man01].

2.14 Signature Padding

The PKCS#1v1.5 [Kal98] padding for a digital signature on message m is

```
EB = 00 || 01 || FF ...  FF || 00 || ASN.1 || H(m)
```

where `ASN.1` is a sequence of bytes that encodes an OID (object identifier string) that describes the hash function and signature algorithm, encoded using ASN.1, and H is a hash function like SHA-256. The textbook RSA signing procedure is applied to the padded value `EB`. Some variants of PKCS#1v1.5 signature padding may be secure in some models [JKM18].

PKCS#1v1.5 also specifies a padding type that uses `00` bytes for padding, but we have never seen it used.

The IPsec IKE (Internet Key Exchange) RFC [HC98] specifies that RSA signatures used for IPsec authentication should be encoded as private key decryptions rather than PKCS#1v1.5 signatures, but the implementations we examined appear to use signature padding.

Bleichenbacher's Low-Exponent PKCS#1v1.5 Signature Forgery Bleichenbacher observed in a 2006 Crypto rump session [Fin06] that, for small public exponents, the PKCS#1v1.5 padding function can be vulnerable to a signature forgery attack against implementations that do not check the full length of the `FF ... FF` padding string. Let us specialize to the case $e = 3$. Then to generate a forged signature, the attacker simply needs to find a string

$$c = \texttt{00 || 01 || FF || FF || FF || 00 || ASN.1 || H(m) || G}$$

where $c < N$, `ASN.1` and `H` are as above, and the vaue `G` is chosen so that when the value c is interpreted as the hexadecimal representation of an integer, it is a perfect cube over the integers. Then the attacker can 'forge' a signature s that will validate against these implementations by returning the value $s = c^{1/3}$ over the integers. Lazy implementations that simply begin matching the padding format from the most significant bits of $s^e \bmod N$ without verifying that the length of the padding string `FF ... FF` is correct for the key

length will validate this signature as valid. A simple method that will often succeed in generating a perfect cube is to start with the integer corresponding to

$$b = \texttt{00 || 01 || FF FF FF || 00 || ASN.1 || H(m) || FF ... FF}$$

where the number of trailing bits is chosen so that the the integer corresponding to this value is less than the modulus N. Then the forged signature is $s = \lfloor b^{1/3} \rfloor$, where the cube root is computed over $\mathbb{R}$ and rounded down to the nearest integer. The value c above is s^3.

Numerous RSA signature implementations, including OpenSSL, were vulnerable to this attack in 2006; in 2014 this vulnerability was found in the Mozilla NSS library [DL14].

2.15 Cross-Protocol Attacks

Wagner and Schneier [WS96] found a theoretical protocol vulnerability with the SSL 3.0 specification that could allow an attacker to substitute signed ephemeral RSA and Diffie–Hellman public keys for each other, because the server's digital signature on the keys did not include the key-exchange algorithm. In their attack, a message containing a Diffie–Hellman prime modulus p and generator g would be interpreted as an RSA modulus p with exponent g. If the client encrypts the key exchange message m to this public key by computing $c = m^g \bmod p$, an attacker can easily recover m by computing $c^{1/g} \bmod p$, since p is prime. Although it was not ruled out by the specification, implementations at the time apparently did not allow this attack in practice.

A 2015 vulnerability in JSON Web Token libraries allowed attackers to forge authentication tokens because the libraries did not tie the algorithm type to the verification key, and used both public and private-key verification algorithms. In the attack, libraries could be confused into using an RSA public key as an HMAC secret key [McL15].

2.16 Key Theft and Cryptanalysis

When RSA is used as a key-exchange or key-encapsulation mechanism, the same long-term public RSA key is typically used to encrypt many messages or sessions. In the context of a protocol like TLS, long-term RSA public keys are validated by using digital signatures from certificate authorities, and certificate validity periods are typically months to years in duration. If an adversary is ever able to steal or mount a cryptanalytic attack against the private key corresponding to one of these long-term RSA public keys, then the adversary would be able to passively decrypt any sessions or messages that had ever used that public key for key exchange.

National Security Agency (NSA) slides leaked by Edward Snowden in 2013 mentioned using known RSA private keys to passively decrypt SSL/TLS network traffic [Sec15] as well as targeted hacking operations to learn pre-shared key values to enable passive decryption of VPN connections [VPN10].

Because of the risk of passive decryption attacks, as well as persistent cryptographic vulnerabilities resulting from PKCS#1v1.5 padding, TLS 1.3 removed RSA key exchange entirely from the protocol, allowing only elliptic-curve or prime-field Diffie–Hellman key exchange [Gil16].

There have also been multiple documented instances of attackers compromising certificate authority digital signing infrastructure to issue fraudulent certificates, including the Comodo [Com11] and DigiNotar [Adk11] hacks in 2011 that enabled man-in-the-middle attacks against Google in Iran. Browser vendors responded to these compromises by adopting public-key pinning, which ties the hash of a public key to its associated domains [Moz17], and certificate transparency [LLK13], in which all valid issued certificates are published in a tamperproof log to provide defense in depth against the risk of signing key compromise.

2.17 Countermeasures

The most straightforward countermeasure against the attacks discussed in this section is to avoid using RSA entirely. For key-exchange or key encapsulation, elliptic-curve cryptography offers smaller key sizes and more efficient operation. Eliminating RSA can be difficult for legacy protocols, unfortunately: there is evidence that the adoption of TLS 1.3 has been delayed because of the decision to eliminate RSA key exchange. If RSA must be used, then Shoup's RSA-KEM scheme [Sho01] sidesteps padding implementation issues entirely: one uses RSA to encrypt a random string of the same length as the key, and derives a symmetric key by applying a key derivation function to this random message. RSA signatures may still be more efficient to verify than elliptic-curve signatures despite their increased size; in this case, a padding scheme like RSA-PSS [BR96] was designed to be provably secure.

3 Diffie–Hellman

Diffie–Hellman key exchange is a required step in many network protocols, including SSH [YL06], the IPsec IKE handshake [HC98, KHN+14], and TLS 1.3 [Gil16]. It has also been an optional key-exchange method in SSL/TLS since SSLv3, and a cipher suite including Diffie–Hellman key exchange was required by the TLS 1.0 specification [DA99]. In this section, we focus on Diffie–Hellman over prime fields, which was historically the only option, and is still supported by the protocols we list above. In the past five years, elliptic-curve Diffie–Hellman (ECDH) has replaced RSA key exchange and prime-field Diffie–Hellman as the most popular key-exchange method; we cover ECDH in Section 4.

3.1 Diffie–Hellman Key Exchange

Diffie–Hellman, in Theory Textbook prime-field Diffie–Hellman key exchange works as follows [DH76]: first, the two parties agree somehow on a prime p and a generator g of a multiplicative group mod$p$. To carry out the key exchange, Alice generates a secret integer exponent a, and sends the value $y_a = g^a \bmod p$ to Bob. Bob responds with the value $y_b = g^b \bmod p$. Alice computes the value $y_b^a = g^{ab} \bmod p$, and Bob computes the value $y_a^b = g^{ba} \bmod p = g^{ab} \bmod p$, so they have a shared value.

The most straightforward means of attack for an attacker is to compute the discrete logarithm of one of the key-exchange messages, although this is not known to be equivalent to computing the Diffie–Hellman shared secret [Mau94].

Diffie–Hellman, in Practice The textbook description of Diffie–Hellman leaves a wide number of choices to implementers, including the type of group, how groups are agreed on, and exponent generation. Different standards and recommendations differ on all of these choices, and implementations differ further from standards.

In the context of protocols, symmetric-key material is typically computed by applying a key derivation function to the Diffie–Hellman shared secret together with other messages from the client–server handshake, and digital signatures are used to authenticate the key exchange against man-in-the-middle attacks. The details of authentication vary across protocols, resulting in a different set of vulnerabilities for different protocols: in SSH [YL06] and TLS 1.3 [Res18], the server digitally signs the entire handshake; in TLS versions 1.2 [DR08] and below, the server digitally signs the Diffie–Hellman key-exchange values and handshake nonces, and the handshake is authenticated by using the derived symmetric keys. IPsec offers numerous authentication options negotiated as part of the cipher suite [HC98, KHN+14].

ElGamal Encryption ElGamal public-key encryption is not commonly supported among most of the network protocols we discuss in this chapter, although it was historically a popular option for PGP. Lenstra *et al.* [LHA+12] report that 47% of the PGP public keys in a public repository in 2012 were ElGamal public keys. Because the structure of ElGamal is very close to Diffie–Hellman, and thus the implementation issues are related, we include it in this section; we briefly remark on its security in Section 3.8.

Textbook ElGamal encryption works as follows [ElG84]. An ElGamal public key contains several parameters specifying the group to be used: a group generator g, a modulus p, and the order q of the subgroup generated by g. Alice's private key is a secret exponent a, and the public key is a value $h_a = g^a \bmod p$. To encrypt a message m to Alice's public key, Bob chooses a secret exponent b and computes the values $h_b = g^b \bmod p$ and $h_{ab} = h_a^b \bmod p$. The ciphertext is the pair $(h_b, m \cdot h_{ab} \bmod p)$. To decrypt, Alice computes the value $h_{ab} = h_b^a \bmod p$ and multiplies the second element of the ciphertext by h_{ab}^{-1} to recover m.

3.2 Group Agreement

Before carrying out the key exchange, the two parties must agree on the group parameters p and g. Different protocols do this differently. In TLS versions 1.2 and earlier [DR08], the server generates the group parameters and sends them to the client together with the server's key exchange message. The server signs this key-exchange message and the random handshake nonces using its long-term public key to prevent man-in-the-middle attacks. In TLS version 1.3 [Res18] servers no longer generate their own Diffie–Hellman group parameters. Instead, the client and server negotiate a group choice from a pre-specified list of Diffie–Hellman groups. The prime-field groups were custom-generated for TLS 1.3 [Gil16]. The IKE key-exhange for the IPsec protocol [HC98, KHN+14] specifies a pre-defined list of Diffie–Hellman groups, and the client and server negotiate the choice of group while negotiating cipher suites. SSH [YL06] includes a few pre-generated groups in the specification, but also allows 'negotiated' groups [FPS06], in which the client specifies their desired prime length, and the server responds with one of a custom list of server-chosen group parameters of the desired size.

These differences in how groups are specified and agreed upon mean that the impact of the Diffie–Hellman vulnerabilities discussed below can look very different for these three network protocols, and the threat model is different for clients and servers even though textbook Diffie–Hellman appears to be equally contributory from both parties. The standardised groups chosen for TLS 1.3, IKE, and SSH were generated to avoid some of the vulnerabilities discussed below. There is no vetting process, and no feasible way for clients to evaluate the quality of custom server-generated Diffie–Hellman parameters for SSH or TLS v1.2 and below. The distribution of groups used for these different protocols also looks very different: for servers supporting TLS v1.2 and earlier, there are a handful of common values and a long tail of custom values, while most IKE servers prefer the same group parameters [ABD+15].

3.3 Prime Lengths

The number field sieve algorithms for factoring and discrete logarithm have the same asymptotic running times, but the discrete logarithm problem is believed to be slightly harder in practice [LV01], and discrete logarithm records have historically lagged several years behind factoring records of the same length. While 512-bit RSA factorisation was first carried out in 1999, the first discrete logarithm computation exceeding this, at 530 bits, dates to 2007 [Kle07]. There is a similar gap between the 768-bit factoring and discrete log records: 2009 for factoring [KAF+10], and 2016 for discrete log [KDL+17]. Nevertheless, the official key-size recommendations for prime-field Diffie–Hellman have historically been

quite similar to those for RSA [LV01]. The use of 1024-bit Diffie–Hellman moduli was allowed by NIST until 2010, deprecated until 2013, and disallowed after 2013 [BR11].

Despite these recommendations, 512-bit and 1024-bit Diffie–Hellman remained in common use well after 2010. In 2015, 8% of popular HTTPS web sites still supported 512-bit 'export'-grade Diffie–Hellman cipher suites [ABD+15], even though server maintainers had already been encouraged earlier that year to disable TLS export cipher suites in the wake of the publicity around the FREAK downgrade attack against export-grade RSA [BBD+15].

Historically, 1024-bit primes for Diffie–Hellman have been very common. In 2015, 91% of IPsec servers supported 1024-bit primes and 66% of them preferred 1024-bit primes over other options in common client configurations; 99% of SSH servers supported 1024-bit primes and 26% of them preferred 1024-bit primes over other options in common client configurations; and 84% of HTTPS servers supporting prime-field Diffie–Hellman used a 1024-bit prime for key exchange [ABD+15]. In 2017, 95% of HTTPS Diffie–Hellman connections seen by Google Chrome telemetry used 1024-bit primes [Chr17].

Software maintainers had difficulty increasing Diffie–Hellman key sizes due to interoperability issues stemming from hard-coded size limits in libraries and hard-coded 1024-bit primes in libraries and specifications. Java JDK versions prior to version 8 did not support primes larger than 1024 bits for Diffie–Hellman key exchange and DSA signatures [Ora14]. Servers using Java without further upgrades could not generate or use larger Diffie–Hellman keys, and older Java-based clients could not handle a larger modulus presented by a server. The SSHv2 transport layer specification [YL06] includes a hard-coded 1024-bit prime group for Diffie–Hellman key exchange, the Oakley Group 2 discussed in Section 3.4, which all implementations were required to support. This group was removed from default support by OpenSSH in 2015, but allowed as a 'legacy' option for backward compatibility [Ope20]. Many major web browsers raised the minimum Diffie–Hellman modulus length for HTTPS to 768 or 1024 bits in 2015 [Res15], but by 2017, it appeared to be easier for browsers to remove support entirely for prime-field Diffie–Hellman for old versions of TLS in favour of elliptic-curve Diffie–Hellman than to increase minimum key strengths to 2048 bits [Chr17, tra15]. This issue has been avoided in TLS 1.3 [Gil16]: the protocol supports only fixed groups, and the shortest Diffie–Hellman prime included in the standard is 2048 bits.

3.4 Standardised and Hard-Coded Primes

A number of protocols and implementations have pre-generated primes with various properties for use in Diffie–Hellman. Some of the most widely used primes have been carefully generated to ensure various desirable properties; the provenance of others is less well documented although the group structures are verifiable.

Some of the most commonly used primes for Diffie–Hellman on the Internet originated with RFC 2412, the Oakley key determination protocol [Orm98], which specified three 'mod p' groups and two elliptic-curve groups. These primes have the property that the high and low 64 bits are all clamped to 1, with the explanation that this helps remaindering algorithms [Orm98]. The middle bits are the binary expansion of π, intended to be a 'nothing up my sleeve' number to allay concerns of trapdoors. The middle bits are incremented until the prime is a 'safe' prime, so $p = 2q + 1$ for prime q, and 2 generates the subgroup of order q. These primes were built into the IKEv1 [HC98], IKEv2 [KHN+14] and SSH [YL06] key exchange protocols as named groups that should be supported by implementations and that can be negotiated for Diffie–Hellman key exchange, and the 1024-bit and 2048-bit primes of this form were historically some of the most commonly used values for Diffie–Hellman key exchange for these protocols.

Adrian *et al.* [ABD+15] estimated that, in 2015, 66% of IPsec servers and 26% of SSH servers used the 1024-bit Oakley prime by default for Diffie–Hellman, and 18% of the

Alexa Top 1 Million HTTPS web sites used a 1024-bit prime that was hard-coded into the Apache web server software by default.

The TLS 1.3 protocol includes several named Diffie–Hellman primes that have the same structure as the Oakley groups, except that they use the binary expansion of e instead of π [Gil16].

3.5 Fixed Primes and Pre-Computation

The fact that the number field sieve algorithms for factoring and discrete log have the same asymptotic complexity has led to the heuristic estimate that equal key sizes for RSA and prime-field Diffie–Hellman offer approximately the same amount of bit security.

As discussed previously in Section 3.3, 1024-bit Diffie–Hellman primes remained in common use for many years after 1024-bit number field sieve computations were believed to be tractable for powerful adversaries. Many implementers justified the continued use of 1024-bit Diffie–Hellman via what they believed to be a calculated risk: given the high estimated cost of 1024-bit number field sieve computations, it was thought that even a powerful adversary would likely be able to carry out at most only a handful of such computations per year. RSA keys typically have long validity periods and thus a single factoring computation would allow the adversary to decrypt many RSA messages encrypted to the broken private key, whereas implementations typically generate new Diffie–Hellman secrets per session, and thus in principle an attacker would need to carry out a new discrete log computation for each session. Prior to 2015, practitioners believed that adversaries would be unlikely to expend the high cost of a full 1024-bit number field sieve computation to break a single Diffie–Hellman key exchange, and thus chose to accept the risk of cryptanalytic attacks in order to take advantage of the added efficiency of small keys and avoid the interoperability issues from increasing key sizes.

However, this popular understanding of computational power was incomplete. The most computationally expensive stages of the number field sieve discrete logarithm calculation depend only on the prime modulus, and the final individual logarithm phase of the algorithm that actually computes the log of the target is asymptotically faster [CS06, Bar13]. In practice, this asymptotic difference means that the individual log computation is significantly faster than the prime-dependent precomputation.

This means that a well-resourced adversary could perform a single expensive precomputation for a given 1024-bit prime modulus, after which the private keys for many individual sessions using that prime would be relatively efficient in practice to compute. This attack is rendered more effective in practice because the adversary could target a small number of hard-coded and standardised 1024-bit primes that account for a relatively large fraction of Diffie–Hellman moduli used in the wild [ABD+15]. Adrian *et al.* estimate that the 66% of IPsec servers and 26% of SSH servers that defaulted to the 1024-bit Oakley prime would be vulnerable to passive decryption by an adversary who carried out a single discrete log precomputation for that prime, and that carrying out ten discrete log precomputations would allow passive decryption to 24% of the most popular HTTPS sites in 2015 [ABD+15].

3.6 Short Exponents

It is quite common for implementations to generate 'short' Diffie–Hellman secret exponents by default. In principle for an otherwise secure Diffie–Hellman group, the strongest attack for short exponents is the Pollard lambda algorithm [Pol78], which takes $O(2^{n/2})$ time against an n-bit exponent a. Thus implementations that wish to achieve a 128-bit security level often generate 256-bit exponents. The TLS 1.3 specification [Gil16] as well as the SSH group exchange specification [FPS06] both suggest the use of shorter exponents for performance reasons, as long as they are at least twice the length of the derived secret.

Valenta *et al.* [VAS+17] examined nine different TLS libraries implementing Diffie–Hellman in 2016 and found that eight of the nine used short exponents. Of these libraries, the Mozilla NSS and libTomCrypt libraries used hardcoded short exponent lengths, the Java OpenJDK uses the max of $n/2$ and 384 for an n-bit prime, OpenSSL and GnuTLS used the bit length of the subgroup order (if specified), with a max of 256 in the latter case, and three libraries used a quadratic curve or work factor calculation to set the bit length of the exponent to match the expected cryptanalytic complexity of the number field sieve algorithm for the length of the prime.

Short exponent lengths are not a vulnerability on their own, but the use of short exponents in combination with common implementation flaws can make some of the key recovery attacks described in Sections 3.8 and 3.10 more severe.

Some implementations appear to generate pathologically short exponents. In 2017, Joshua Fried found that 3% of the 4.3 million hosts that responded to an IKEv1 IPsec handshake and 1.3% of the 2.5 million hosts that responded to an IKEv2 IPsec handshake used exponents that were shorter than 16 bits. These were found by precomputing 2^{17} public key-exchange values for positive and negative 16-bit exponents for the most common Diffie–Hellman groups, and comparing these to the key-exchange messages transmitted by servers [Fri20].

3.7 Exponent Re-Use

Many implementations reuse Diffie–Hellman exponents by default across multiple connections. OpenSSL reused Diffie–Hellman exponents by default until January 2016, unless a specific `SSL_OP_SINGLE_DH_USE` flag was set. Springall *et al.* found in 2016 that 7% of the Alexa Top 1 Million HTTPS servers who supported prime-field Diffie–Hellman reused Diffie–Hellman key-exchange values [SDH16].

Re-use of Diffie–Hellman key-exchange values for multiple connections is not in principle a vulnerability, and should be no less secure than using an RSA public key for encryption across multiple connections but, in practice, key-exchange message re-use can make some of the attacks described in Section 3.10 more severe.

3.8 Group Structure

For prime-field Diffie–Hellman, there is a variety of choices of primes and group structures that implementations can choose. The security of Diffie–Hellman relies crucially on the structure of the group.

For a prime modulus p, the order of the group generated by g will divide $p-1$. The Pollard rho [Pol74] and 'baby step giant step' algorithms have running times that depend on the group order: for a group of order q, these algorithms run in time $O(\sqrt{q})$.

If the order of the group generated by g has a prime factor q_i, then an adversary can take advantage of the existence of a subgroup of order q_i, and use one of these algorithms to compute the discrete logarithm of a target modulo q_i. If the order of the group generated by g has many such subgroups, the attacker can use the Pohlig–Hellman algorithm [PH78] to repeat this for many subgroup orders q_i and use the Chinese remainder theorem and Hensel lifting to compute the secret exponent a modulo the product of the known q_i.

To protect against these attacks, implementations should choose g such that g generates a subgroup of large prime-order q, where q should have bit length at least twice the desired security parameter of the encryption.

Safe Primes A common recommendation is for implementations to use 'safe' primes, where $p = 2q + 1$ for q a prime, and then to use a generator g of the subgroup of order q modulo p. This protects against attacks based on subgroup structure by maximising the order of the subgroup an attacker would need to attack.

Until 2019, OpenSSL would by default generate Diffie–Hellman group parameters where p was a 'safe' prime, but where g generated the 'full' group of order $2q$ modulo p [Edl19]. This meant that an adversary could always compute one bit of information about the exponent in any key-exchange message, by computing the discrete log in the subgroup of order 2. Of around 70 000 distinct group parameters g, p in use by HTTPS servers in 2015, around 64 000 of the prime moduli p were 'safe', and only 1250 of those used a generator g that generated a group of prime-order q [ABD+15]. In other words, in practice the Decisional Diffie–Hellman assumption [Bon98] is often false. In particular, textbook ElGamal encryption, where a message is chosen as a non-zero integer modulo p, is not semantically secure as a consequence.

DSA-Style Groups An alternative common structure for primes used for Diffie–Hellman key exchange is to use a prime p generated so that $p-1$ has a prime factor q of a fixed size much smaller than p, and g is chosen in order to generate the subgroup of order q modulo p. Commonly encountered parameter lengths include choosing the subgroup order q to be 160 bits for a 1024-bit modulus p, or choosing the subgroup order q to be 224 or 256 bits for a 2048-bit prime p [Inf13]. Groups of this type were originally used for DSA, and then recommended for many years for use in Diffie–Hellman by NIST SP800-56A [BCR+18].

RFC 5114 specified several pre-generated groups of this form for use in SMIME, SSH, TLS and IPsec to conform to the NIST standard [LK08]. In 2017, Valenta *et al.* [VAS+17] found that 11% of the 10.8 million HTTPS servers that supported prime-field Diffie–Hellman were using the 1024-bit Group 22 specified in RFC 5114. In 2018, NIST updated their recommendation to allow only 'safe' primes for Diffie–Hellman. 'DSA'-style groups are now only permitted for backwards compatibility [BCR+18].

In the context of the DSA digital signature algorithm, the presence of these subgroups permits shorter digital signatures, and allows implementations to use hash functions of common lengths. The length of the subgroup is chosen so that the bit complexity of different families of attacks matches the desired overall security of the system [Len01]: for the 1024-bit parameters, the number field sieve is believed to take about 2^{80} time, which matches the expected complexity of carrying out a Pollard rho attack against the 160-bit subgroup q, or the Pollard lambda attack against the 160-bit secret exponent.

The reasoning behind the recommendation to use smaller subgroups appears to have been a desire to use short secret exponents to make the modular exponentiations required for key exchange more efficient, combined with concerns about vulnerability to hypothetical attacks if the exponents were much shorter than the subgroup order. Such attacks exist for DSA signatures [HGS01], but this is not known to be a vulnerability in the Diffie–Hellman setting. However, the use of smaller subgroups necessitates additional validity checks to prevent small subgroup attacks, described in Section 3.10.

Unstructured Primes and Composite-Order Groups For an arbitrary randomly generated prime p, the prime factorisation of $(p-1)/2$ is expected to look like a random integer. That is, it is likely to have many small factors, some medium-sized factors, and a few large factors. The expected length of the largest subgroup order for a 1024-bit prime is over 600 bits [RGB+16], meaning that using the full Pohlig–Hellman algorithm [PH78] to compute arbitrary discrete logarithms modulo such a prime is likely to be infeasible due to the square root running time of computing discrete logs in such a large subgroup.

However, if the target of the attack also uses short exponents, van Oorschot and Wiener's attack [vW96] exploits the fact that an attacker could combine Pollard's lambda algorithm [Pol78] with the Pohlig–Hellman algorithm over a strategically chosen subgroup whose order has relatively small prime factors to uniquely recover a shorter exponent. An adversary would be able to recover a 160-bit exponent with 2^{40} computation for 32% of 1024-bit primes.

The prescribed subgroup structures described above are intended to prevent such attacks, but some relatively rare implementations do not appear to take any steps to generate primes with a cryptographically secure structure.

In 2015, among around 70 000 distinct primes p used for Diffie–Hellman key exchange for 3.4 million HTTPS servers supporting Diffie–Hellman key exchange on the Internet, there were 750 groups for which $(p-1)/2$ was not prime, and an opportunistic effort to factor $(p-1)/2$ using the ECM algorithm [Len87] revealed prime factors of the order of g. This allowed Adrian *et al.* [ABD+15] to apply van Oorschot and Wiener's attack to compute the full exponent for 159 key exchanges (many of which were using 128-bit exponents) and partial information for 460 key exchanges.

Composite Moduli Diffie–Hellman key exchange is almost universally described as being carried out modulo a prime p. However, non-prime moduli have been found in the wild.

If a non-prime Diffie–Hellman modulus can be efficiently factored, then, in general, computing the discrete logarithm, and thus the private key for a Diffie–Hellman key exchange, is only approximately as difficult as the problem of factoring the modulus and computing the discrete logarithm of the target key-exchange message for each prime factor or subgroup of the modulus. The Chinese remainder theorem can then be used to reconstruct the discrete logarithm modulo the least common multiple of the totient function of each of the prime factors.

Not all non-prime Diffie–Hellman moduli are necessarily insecure: a hard-to-factor composite RSA modulus, for example, would likely still be secure to use for Diffie–Hellman, since an adversary would not be able to factor the modulus to learn the group structure. A composite Diffie–Hellman modulus with one small factor where computing the discrete logarithm is easy and one large factor where computing the discrete logarithm is still difficult could allow an adversary to compute partial information about the secret exponent. And a highly composite Diffie–Hellman modulus where every prime factor is relatively small or admits an easy discrete logarithm would allow the adversary to efficiently compute the full secret exponent.

In 2016, the Socat tool was found to be using a hard-coded non-prime modulus p of unknown origin for Diffie–Hellman key exchange [Rie16]. Lenstra *et al.* [LHA+12] found 82 ElGamal public keys in the PGP key repository in 2012 using non-prime p values. Many of these shared bit patterns with other group parameters used by PGP users, suggesting they may have been invalid, corrupted keys.

Valenta *et al.* [VAS+17] found 717 SMTP servers in 2016 in the wild using a 512-bit composite Diffie–Hellman modulus whose hexadecimal representation differed in one byte to a default Diffie–Hellman prime included in SSLeay, the predecessor to OpenSSL.

3.9 Group Parameter Confusion

Adrian *et al.* [ABD+15] found 5700 HTTPS hosts that were using DSA-style group parameters that had been hard-coded in Java's `sun.security.provider` package for Diffie–Hellman, except that they were using the group order q in place of the generator g. Adrian *et al.* hypothesised that this stemmed from a usability problem. The ASN.1 representation of Diffie–Hellman key-exchange parameters in PKCS#3 is the sequence (p, g), while DSA parameters are specified as (p, q, g). For the parameters with 512-bit p, the group generated by q would leak 290 bits of information about the secret exponent using 2^{40} computation. Java uses 384-bit exponents for this prime length. Computing discrete logarithms for this group would thus be more efficient using the number field sieve than using Pollard lambda on the remaining bits of the exponent, but this is a near-miss vulnerability.

3.10 Small Subgroup Attacks

Diffie–Hellman is vulnerable to a variety of attacks in which one party sends a maliciously crafted key-exchange value that reduces the security of the shared secret by confining it to a small set of values. These attacks could allow a man-in-the-middle attacker to coerce a Diffie–Hellman shared secret to an insecure value, or let an attacker learn information about a victim's secret exponent by carrying out a protocol handshake with them.

Zero-Confinement For prime-field Diffie–Hellman, the value 0 is not a member of the multiplicative group modulo p, and therefore should never be sent as a key-exchange value. However, if Alice sends the value 0 as her key exchange value to Bob, then Bob will derive the value 0 as his shared secret. For protocols like TLS versions 1.2 and below, where the integrity of the handshake is ensured using a key derived from the shared secret, then this could allow a man-in-the-middle to compromise the security of the key exchange. To protect against this type of attack, implementations must reject the value 0 as a key-exchange value.

Valenta *et al.* [VAS+17] scanned the Internet in 2016 and found that around 3% of SSH servers and 0.06% of HTTPS servers were willing to accept a Diffie–Hellman key-exchange value of 0. They note that, until 2015, a vulnerability in the Libreswan and Openswan IPsec VPN implementations caused the daemon to restart when it received 0 as a key exchange value [CVE15].

Subgroups of Order 1 and 2 Since $(p-1)$ is even for any prime $p > 2$, there will be a multiplicative subgroup of order 2 modulo p, generated by the value -1, as well as the trivial group of order 1 generated by the value 1. For Diffie–Hellman, if g generates the full group of integers modulo p, then the group of order 2 generated by -1 will be a proper subgroup; if g is a 'safe' prime or a DSA-style prime as described above and g is chosen to generate a subgroup of prime-order q, then the value -1 will not be contained in the group generated by g. The value 1 is contained in the group in either case.

Thus, similar to the case of 0, Alice could send the element 1 as a key exchange value and ensure that the resulting shared secret derived by Bob is the value 1, or send -1 and ensure that Bob's resulting secret is either 1 or -1. In the latter case, if Alice subsequently learns a value derived from Bob's view of the shared secret (for example, a ciphertext or MAC, depending on the protocol) then Alice can learn one bit of information about Bob's secret exponent, whether or not the Diffie–Hellman group Bob intended to use had large prime order.

To prevent these attacks, implementations must reject the values 1 and -1 as key-exchange values.

Valenta *et al.* [VAS+17] report that, in 2016, 3% of HTTPS servers and 25% of SSH servers accepted the value 1, and 5% of HTTPS servers and 34% of SSH servers accepted the value -1 as a Diffie–Hellman key-exchange value.

Subgroups of Larger Order For 'safe' primes p, the validation checks eliminating the values 0, 1 and -1 are the only checks necessary to eliminate small subgroup confinement attacks. For DSA-style primes where g generates a group of order q where q is much less than p, then there are subgroups modulo p for each of the factors of $(p-1)/q$. The recommended prime-generation procedures do not require the co-factor $(p-1)/(2q)$ to be prime, so in practice many primes in use have co-factors that are random integers with many small prime factors. For example, the 1024-bit prime p specified in RFC 5114 [LK08] was chosen to have a 160-bit prime-order subgroup, but it also has a subgroup of order 7.

Let g_7 be a generator of the group of order 7 modulo p. If Alice sends g_7 as her key-exchange value, then the resulting shared secret derived by Bob will be confined to this subgroup of order 7. If Alice subsequently learns a value derived from Bob's view of

the shared secret, then Alice can compute Bob's secret exponent $b \bmod 7$ by brute forcing over the subgroup size.

Lim and Lee [LL97] developed a full secret key recovery attack based on these principles. In the Lim–Lee attack, the victim Bob reuses the secret exponent for multiple connections, and the attacker Alice wishes to recover this secret. Alice finds many small subgroups of order q_i modulo Bob's choice of Diffie–Hellman prime p, sends generators g_{q_i} of each subgroup order in sequence, and receives a value derived from Bob's view of the secret shared key in return. Alice then uses this information to recover Bob's secret exponent $b \bmod q_i$ for each q_i. Depending on the protocol, this may take $O(\sum_i \sqrt{(q_i)})$ time if Bob were to directly send back his view of the key share $g_{q_i}^b$, or $O(\sum_i q_i)$ time if Bob sends back a ciphertext or MAC whose secret key is derived from Bob's key share $g_{q_i}^b$.

To prevent these small subgroup attacks when using groups of this form, implementations must validate that received key-exchange values y are in the correct subgroup of order q by checking that $y^q \equiv 1 \bmod p$. Unfortunately, although TLS versions 1.2 and below and SSH allow servers to generate their own Diffie–Hellman groups, the data structures used for transmitting those groups to the client do not include a field to specify the subgroup order, so it is not possible for clients to perform these validation checks. In principle, it should be feasible for servers to perform these validation checks on key-exchange values received by clients, but in practice, Valenta *et al.* found in 2016 that almost no servers in the wild actually performed these checks [VAS+17].

In 2016, OpenSSL's implementation of the RFC 5114 primes was vulnerable to a full Lim–Lee key-recovery attack, because it failed to validate that Diffie–Hellman key-exchange values were contained in the correct subgroup, and servers reused exponents by default. For the 2048-bit prime with a 224-bit subgroup specified in RFC 5114, a full key recovery attack would require 2^{33} online work and 2^{47} offline work [VAS+17].

These attacks could also have been mitigated by requiring the co-factor of these groups to have no small factors, so that primes would have the form $p = 2qh + 1$, where the co-factor h is prime or has no factors smaller than the subgroup order q [LL97]. This was not suggested by the relevant recommendations [BCR+18].

3.11 Cross-Protocol Attacks

In TLS versions 1.2 and earlier, the server signs its Diffie–Hellman key exchange message to prevent man-in-the-middle attacks. However, the signed message does not include the specific cipher suite negotiated by the two parties, which enables multiple types of cross-protocol attacks.

A simple case is the Logjam attack of Adrian *et al.* [ABD+15]. In this attack, a man-in-the-middle attacker impersonates the client, negotiates an 'export-grade' Diffie–Hellman cipher suite with the server, and receives the server's Diffie–Hellman key-exchange message using a 512-bit prime, signed with the server's certificate private key. This message does not include any indication that it should be bound to an 'export-grade' Diffie–Hellman cipher suite. The attacker can then impersonate the server and forward this weak key-exchange message to the victim client, who believes that they have negotiated a normal-strength Diffie–Hellman key-exchange with the server. From the client's perspective, this message is indistinguishable from a server who always uses a 512-bit prime for Diffie–Hellman. In order to be successful in completing the man-in-the-middle attack undetected, the attacker must compute the 512-bit discrete log of the client or server's key-exchange message in order to compute the symmetric authentication keys, forge the client and server MAC contained in the 'Finished' messages sent at the end of the handshake, and decrypt the symmetrically encrypted data. This was feasible or close to feasible, in practice, for 512-bit primes because of the benefits of precomputation, discussed previously in Section 3.5.

Mavrogiannopoulos *et al.* [MVVP12] observed that TLS versions 1.2 and prior are vulnerable to a more sophisticated cross-cipher-suite attack in which an attacker convinces

the victim to interpret a signed elliptic-curve Diffie–Hellman key-exchange message as a valid prime-field Diffie–Hellman key-exchange message. This attack exploits three protocol features: first, for prime-field Diffie–Hellman, the server includes its choice of prime and group generator as explicit parameters along with the ephemeral Diffie–Hellman key-exchange value; second, for elliptic-curve Diffie–Hellman, the key-exchange message included an option to specify an explicit elliptic-curve via the curve parameters; and third, these messages do not include an indication of the server's choice of cipher suite.

Putting these properties together for an attack, the man-in-the-middle impersonates the client and initiates many connections to the server requesting an ECDH cipher suite until it receives an ECDH key-exchange message that can be parsed as a prime-field Diffie–Hellman key-exchange message by using a weak modulus. At this point, the man-in-the-middle attacker forwards this weak message to the victim client, who believes that they have negotiated a prime-field Diffie–Hellman key exchange. The attacker must compute the discrete logarithm online in order to forge the 'Finished' messages and complete the handshake. The key to the attack is that the random ECDH key-exchange values are parsed as a 'random'-looking modulus and group generator for prime-field Diffie–Hellman, which could be exploited as described for the composite-order groups in Section 3.8 above. Mavrogiannopoulos *et al.* estimate that the attacker is likely to succeed after 2^{40} connection attempts with the server.

3.12 Nearby Primes

Many of the most commonly used prime moduli used for Diffie–Hellman share many bits in common with each other, because they have the same fixed most and least significant bits, and use the digits of π or e for the middle bits.

In addition to the fixed, named groups of this form included in the SSH specification, SSH also allows the client and server to negotiate group parameters by specifying approved bit lengths for keys [FPS06]. When this group negotiation is carried out, SSH servers choose group parameters from a pre-generated file (e.g., `/etc/ssh/moduli` on Linux) of primes and group generators. Default files of these parameters are distributed along with the operating system.

These files contain dozens of primes of each specified key length that all differ in only the least-significant handful of bits. They appear to have been generated by starting at a given starting point, and incrementing to output 'safe' primes.

It is not known whether an adversary would be able amortise cryptanalytic attacks over many nearby primes of this form via the number field sieve or other discrete logarithm algorithms.

3.13 Special Number Field Sieve

The number field sieve (NFS) is particularly efficient for integers that have a special form. This algorithm is known as the special number field sieve (SNFS). The improved running time applies for integers p where the attacker knows a pair of polynomials f, g, of relatively low degree and small coefficients, such that they share a common root modulo p. A simple case would be a p of the form $m^6 + c$ for small c; this results in the polynomial pair $f(x) = x^6 + c$ and $g(x) = x - m$.

Although neither a general 1024-bit factorisation nor discrete logarithm have been known to have been carried out in public as of this writing (2020), a 1039-bit SNFS factorisation was completed in 2007 by Aoki *et al.* [AFK+07], and a 1024-bit SNFS discrete logarithm was carried out in 2016 by Fried *et al.* in a few calendar months [FGHT17].

The latter note that they found several Diffie–Hellman primes in the wild with forms that were clearly amenable to SNFS computations. These included 150 HTTPS hosts using the 512-bit prime $2^{512} - 38\,117$, and 170 hosts using the 1024-bit prime $2^{1024} - 1\,093\,337$ in

March 2015. They also report that the LibTomCrypt library included several hard-coded Diffie–Hellman groups ranging in sizes from 768 to 4096 bits, with a readily apparent special form amenable to the SNFS. The justification for using such primes appears to have been to make modular exponentiation more efficient [LL96].

SNFS Trapdoors In 1991, in the context of debates surrounding the standardisation of the Digital Signature Algorithm, Lenstra and Haber [LH91] and Gordon [Gor93] raised the possibility that an adversary could construct a malicious prime with a hidden trapdoor structure that rendered it amenable to the SNFS algorithm, but where the trapdoor structure would be infeasible for an outside observer to discover. This could be accomplished by constructing a pair of SNFS polynomials first, with resultant an appropriately structured prime p, and publishing only p, while keeping the polynomial pair secret. Computing discrete logarithms would then be easy for this group for the attacker, but infeasible for anyone who did not possess the secret. Although this type of trapdoor would be difficult to hide computationally for the 512-bit primes in use in the early 1990s, Fried *et al.* [FGHT17] argue that such a trapdoor would be possible to hide for 1024-bit primes.

In response to these concerns about trapdoor primes, the Digital Signature Standard suggests methods to generate primes for DSA in a 'verifiably random' way, with a published seed value [Inf13]. Alternatively, the use of 'nothing up my sleeve' values such as the digits of π or e for the Oakley [Orm98] and TLS 1.3 [Gil16] primes is intended to certify that these primes are not trapdoored, because it would not be possible for an attacker to have embedded the necessary hidden structures.

The use of 'nothing up my sleeve' numbers is easily verifiable for those primes, but for 'verifiably random' primes generated according to FIPS guidelines, publishing the seeds is optional, and almost none of the primes used for either DSA or Diffie–Hellman in the wild are published with the seeds used to generate them. These include several examples of widely used standardised and hard-coded primes whose generation is undocumented. An example would be the primes included in RFC 5114 [LK08], which were taken from the FIPS 186-2 [Inf00] test vectors, but were published without the seeds used to generate them. These primes were used by 2% of HTTPS hosts and 13% of IPsec hosts in 2016 [FGHT17].

3.14 Countermeasures

The most straightforward countermeasure against the attacks we describe against prime-field Diffie–Hellman is to avoid its use entirely, in favour of elliptic-curve Diffie–Hellman. Given the current state of the art, elliptic-curve Diffie–Hellman appears to be more efficient, permit shorter key lengths, and allow less freedom for implementation errors than prime-field Diffie–Hellman. Most major web browsers have dropped support for ephemeral prime-field Diffie–Hellman cipher suites for TLS 1.2 and below [Chr17, tra15], and elliptic-curve Diffie–Hellman now represents the majority of HTTPS handshakes carried out in the wild.

If prime-field Diffie–Hellman must be supported, then the implementation choices made by TLS 1.3 are a good blueprint for avoiding known attacks: groups should use a fixed, pre-generated 'safe' prime deterministically produced from a 'nothing up my sleeve' number, the minimum acceptable prime length is 2048 bits, and any Diffie–Hellman-based protocol handshake must be authenticated via a digital signature over the full handshake.

4 Elliptic-Curve Diffie–Hellman

Elliptic-curve cryptography offers smaller key sizes and thus more efficient operation compared with factoring and finite-field-based public-key cryptography, because sub-

exponential-time index calculus-type algorithms for the elliptic-curve discrete log problem are not known to exist. While the idea of using elliptic curves for cryptography dates to the 1980s [Mil86, Kob87], adoption of elliptic-curve cryptography has been relatively slow, and elliptic-curve cryptography did not begin to see widespread use in network protocols until after 2010. As of this writing, however, elliptic-curve Diffie–Hellman is used in more than 75% of the HTTPS key exchanges observed by the ICSI Certificate Notary [ICS20].

There were numerous contributing factors to the delays in wide adoption of elliptic-curve cryptography, including concerns about patents, suspicions of the NSA's role in the development of standardised curves, competition from RSA, and the belief among practitioners that elliptic curves were poorly understood compared with more the 'approachable' mathematics of modular exponentiation for RSA and prime-field Diffie–Hellman [KKM11].

The NSA has been actively involved in the development and standardisation efforts for elliptic-curve cryptography, and the original version of the NSA's Suite B algorithm recommendations in 2005 for classified US government communications included only elliptic-curve algorithms for key agreement and digital signatures. Thus it came as quite a surprise to the community when the NSA released an announcement in 2015 that owing to a 'transition to quantum resistant algorithms in the not too distant future', those 'that have not yet made the transition to Suite B elliptic-curve algorithms' were recommended to not make 'a significant expenditure to do so at this point but instead to prepare for the upcoming quantum resistant algorithm transition' [Nat15]. The updated recommendations included 3072-bit RSA and prime-field Diffie–Hellman; the algorithm suite has been renamed but the recommended algorithms and key sizes remained the same. Koblitz and Menezes evaluate the suspicions of an algorithmic break or backdoor that this announcement sparked [KM16].

ECDH, in Theory An elliptic-curve group is defined by a set of domain parameters: the order q of the field $\mathbb{F}_q$, the coefficients of the curve equation $y^2 = x^3 + ax + b \bmod q$, and a generator G of a subgroup of prime-order n on the curve. The co-factor h is equal to the number of curve points divided by n.

Alice and Bob carry out an elliptic-curve Diffie–Hellman key exchange in theory as follows: Alice generates a secret exponent k_a and sends the public value $Q_a = k_a G$. Bob generates a secret exponent b and sends the public value $Q_b = k_b G$. Alice then computes the shared secret as $k_a Q_b$, and Bob computes the shared secret as $k_b Q_a$.

Elliptic-curve public keys can be represented in uncompressed form by providing both the x and y coordinates of the public point, or in compressed form by providing the x coordinate and a bit to specify the y value.

4.1 Standardised Curves

In contrast to the situation with prime-field Diffie–Hellman, end users almost never generate their own elliptic curves, and instead rely on collections of standardised curve parameters. Several collections of curve parameters have been published. The SEC 2 [SEC00] recommendation published by Certicom includes 'verifiably random' and Koblitz curve parameters for 192-, 224-, 256-, 384- and 521-bit prime field sizes, and 163-, 233-, 239-, 283-, 409- and 571-bit binary field sizes. NIST included the 224-, 256-, 384- and 521-bit random curves in their elliptic-curve recommendations; the 256-bit prime-field 'verifiably random' NIST P-256 curve is by far the most commonly used one in practice [VAS+17]. The 'verifiably random' curve generation procedure has been criticized for simply hashing opaquely specified values [BCC+15]. Although the likelihood of undetectably backdoored standardised curves is believed to be small [KM16], distrust of standardised elliptic curves has slowed the adoption of elliptic-curve cryptography more generally.

Another prominent family of standardised curves are the Brainpool curves authored by the German ECC Brainpool consortium [LM10], which give 'verifiably pseudorandom'

curves over prime fields of 160-, 192-, 224-, 256-, 320-, 384- and 512-bit lengths. These curves are supported by many cryptographic libraries, although are less popular than the NIST curves [VAS+17].

Curve25519 [Ber06] is a 256-bit curve that was developed by Bernstein to avoid numerous usability and security issues that affect the SEC 2/NIST and Brainpool curves, and has been growing in popularity in network protocols.

TLS versions 1.2 and below included a mechanism by which individual servers could configure custom elliptic-curve parameters, but this does not seem to be used at all for HTTPS servers reachable on the public Internet [VAS+17].

4.2 Invalid Curve Attacks

In principle, a small subgroup attack analogous to the small subgroup attacks described in Section 3.10 for prime-field Diffie–Hellman is also possible against elliptic curves. To mitigate these attacks, the elliptic curves that have been standardised for cryptography have typically been chosen to have small co-factors. NIST recommends a maximum co-factor for various curve sizes.

However, there is a much more severe variant of this attack that is due to Antipa *et al.* [ABM+03]: an attacker could send an elliptic-curve point of small order q_i that lies on a different curve entirely. If the victim does not verify that the received key-exchange value lies on the correct curve, the attacker can use the victim's response to compute the victim's secret modulo q_i. If the victim is using static Diffie–Hellman secrets, then the attacker can repeat this for many chosen curves and curve points of small prime-order to recover the full secret.

To mitigate this attack, implementations must validate that the points it receives lie on the correct curve, or use a scalar multiplication algorithm that computes only on the x-coordinate together with a curve that is secure against curve twist attacks.

Elliptic-curve implementations have suffered from repeated vulnerabilities due to failure to validate that elliptic-curve points are on the correct curve. Jager *et al.* [JSS15] found that three out of eight popular TLS libraries did not validate elliptic-curve points in 2015. Valenta *et al.* [VAS+17] scanned TLS, SSH and IKE addresses in 2016 using a point of small order on an invalid curve and estimated 0.8% of HTTPS hosts and 10% of IKEv2 hosts did not validate ECDH key-exchange messages.

4.3 Countermeasures

Elliptic-curve Diffie–Hellman has been growing in popularity, and currently appears to offer the best security and performance for key exchange in modern network protocols. Compared with the long history of implementation messes involving RSA and prime-field Diffie–Hellman, ECDH seems relatively unscathed. The one dark spot is the relatively expensive validation checks required to protect against the invalid curve and twist attacks that have plagued some implementations of the NIST curves. As a countermeasure, Curve25519 was designed to require only a minimal set of validation checks.

Elliptic-curve Diffie–Hellman (ECDH) implementations may paradoxically be 'protected' from implementation mistakes by a form of security through obscurity. Because the mathematics of elliptic curves is so much more complex than RSA or prime-field Diffie–Hellman, implementers seem empirically less likely to attempt to design their own curves, or deviate from standard recommendations for secure implementations. Another protective factor may have been the relatively late dates of ECDH adoption. By the time elliptic curves began to be used on any scale in the wild, well after 2010, the cryptographic community had already discovered the analogues of many of the basic cryptographic vulnerabilities that early RSA and Diffie–Hellman implementations suffered from.

5 (EC)DSA

The DSA algorithm was originally standardised in the early 1990s. Prime-field DSA never found widespread use for SSL/TLS (Lenstra *et al.* [LHA+12] report finding only 141 DSA keys out of more than 6 million distinct HTTPS certificates in 2012; nearly all of the rest were RSA). However, the SSH specification requires implementations to support DSA as a public-key format [YL06], and it was almost universally supported by SSH servers until it began to be replaced by ECDSA. DSA public keys were also widely used for PGP: Lenstra *et al.* [LHA+12] report that 46% of 5.4 million PGP public keys they scraped in the wild in 2012 were DSA public keys. The relative popularity of DSA compared with RSA in these different protocols is likely due to the fact that RSA was protected by a patent that did not expire until the year 2000.

In the past handful of years, ECDSA, the elliptic-curve digital signature algorithm, has rapidly grown in popularity. According to the ICSI Certificate Notary, 10% of the HTTPS certificates seen in the wild as of this writing use ECDSA as their signature algorithm [ICS20]. Most of the major cryptocurrencies use ECDSA signatures to authenticate transactions.

DSA Key Generation, in Theory The DSA public parameters include a prime p chosen so that $(p-1)$ is divisible by a smaller prime q of specified length, and g a generator of the subgroup of order q modulo p. To generate a DSA public key, the signer generates a random secret exponent x, computes the value $y = g^x \bmod p$, and publishes the values (p, q, g, y) [Inf13].

DSA Signatures and Verification, in Theory As originally published, the DSA is randomised. To sign the hash of a message $H(m)$, the signer chooses an integer k, which is often called a signature nonce, although it serves as an ephemeral private key. The signer computes the values $r = g^k \bmod p \bmod q$ and $s = k^{-1}(H(m) + xr) \bmod q$, and publishes the values (r, s) [Inf13]. To mitigate vulnerabilities due to random-number generation failures in generating k, many implementations now use 'deterministic' nonce generation, where the nonce value k is generated pseudorandomly and deterministically by applying a pseudorandom function, typically based on HMAC, to the message m and the secret key x [Por13].

To verify a signature (r, s) with a public key (p, q, g, y), the verifier computes the values $w = s^{-1} \bmod q$, $u_1 = H(m)w \bmod q$, and $u_2 = rw \bmod q$. The verifier then verifies that $r = g^{u_1} y^{u_2} \bmod p \bmod q$ [Inf13].

5.1 Distinct Primes

Perhaps in response to the trapdoor prime controversy surrounding the standardisation of DSA described in Section 3.13, custom primes for DSA have been historically more common than custom primes for Diffie–Hellman. Lenstra *et al.* [LHA+12] note that, of the more than 2.5 million DSA public keys in the PGP key database in 2012, only 1900 (0.07%) used the same prime as another public key in the database. This is in contrast to the choices for ElGamal PGP public keys, where 66% of the more than 2.5 million ElGamal public keys used a prime that was used by another key in the database, and there were only 93 such primes that were shared.

5.2 Repeated Public Keys

Repeated public keys are also common across hosts. Heninger *et al.* [HDWH12] documented extensive repetition of DSA public keys across SSH hosts in 2012, with common keys being served by thousands of IP addresses. Many of these were large hosting providers

with presumably shared infrastructure, but they also found evidence of hard-coded keys baked into network device firmware, as well as evidence of random-number generation issues similar to those affecting RSA. In these cases, the owner of a vulnerable key will know the corresponding secret key for any host sharing the same public key, and could thus generate valid signatures for these vulnerable hosts. For keys that have been hard-coded into firmware images or repeated due to poor randomness, the secret keys can be compromised by an attacker who reverse-engineers the implementation.

5.3 ECDSA

ECDSA Key Generation, in Theory The public-domain parameters for an ECDSA public key are the same as for elliptic-curve Diffie–Hellman: for the purposes of this section, the relevant curve parameters are a specification of a finite field $\mathbb{F}$ and an elliptic-curve E, together with a generator G of a subgroup of prime-order q on the curve. The private signing key is an integer x, and the public signature key is the point $Y = xG$ [Inf13].

ECDSA Key Generation, in Practice The NIST P-256 curve is by far the most commonly used curve for ECDSA by SSH and HTTPS servers in the wild. The most common cryptocurrencies including Bitcoin, Ethereum and Ripple all use the curve `secp256k1`, which is a 256-bit Koblitz curve described in SEC 2 [SEC00].

The Ed25519 signature scheme [BDL+12] is a variant of EdDSA, the DSA signature scheme adapted to Edwards curves, specialised to Curve25519. It includes countermeasures against many common implementation vulnerabilities: in particular, signature nonces are defined as being generated deterministically, and it is implemented without secret-dependent branches that might introduce side channels.

ECDSA Signatures and Verification, in Theory To sign a message hash $H(m)$, the signer chooses an integer k, computes the point $(x_r, y_r) = kG$, and outputs $r = x_r$ and $s = k^{-1}(H(m) + xr) \bmod q$. The signature is the pair (r, s) [Inf13]. In 'deterministic' ECDSA, the nonce k is generated pseudorandomly and deterministically by applying an HMAC-based pseudorandom function to the message m and the secret key x [Por13].

To verify a message hash using a public key Y, the verifier computes $(x'_r, y'_r) = hs^{-1}G + rs^{-1}Y$ and verifies that $x'_r \equiv r \bmod q$.

Signature Normalisation ECDSA signatures have the property that the signatures (r, s) and $(r, -s)$ will validate with the same public key. In order to ensure that signatures are not malleable, Bitcoin, Ethereum and Ripple all use 'signature normalisation', which uses the smaller of s and $-s$ for a signature. This is a mitigation against attacks in the cryptocurrency context in which an attacker duplicates a transaction under a different transaction identifier (computed as the deterministic hash over all of the transaction data including the signature) by modifying the signature [Kli17].

5.4 Curve Replacement Attacks

In 2020, the NSA announced a critical security vulnerability in Microsoft's Windows 10 certificate validation code [Nat20]. The code flaw failed to validate that the elliptic-curve group parameters in the signature match the curve parameters in a trusted certificate. This allowed an adversary to forge a signature that would validate for any given key under this code by giving the public-key value Y in place of the generator G of the curve, and generating a signature using the secret key $x = 1$.

5.5 Small Secret Keys

Because the security of (EC)DSA relies on the (elliptic-curve) discrete logarithm of the public key y or Y being hard to compute, the secret key x should be difficult to guess. However, numerous implementation vulnerabilities appear to have led to predictable secret exponents being used for ECDSA public keys in the context of Bitcoin. The Large Bitcoin Collider is a project that is using a linear brute-force search algorithm to search for Bitcoin keys with a small public exponent, and has found several secret keys used in the wild after searching a 55-bit space [ric16].

5.6 Predictable Secret Keys

A Bitcoin 'brainwallet' is a tool that derives an ECDSA public–private key pair from a passphrase provided by a user. A typical key derivation might apply a cryptographic hash function H to a passphrase to obtain the ECDSA secret key x. This allows the user to use this public–private key pair without having to store an opaque blob of cryptographic parameters in a secure fashion. However, attackers have carried out dictionary attacks and successfully recovered numerous Bitcoin secret keys, together with associated funds [Cas15].

5.7 Predictable Nonces

The security of (EC)DSA also relies crucially on the signature nonce (actually a one-time secret key) for every signature remaining secret and hard to predict. If an attacker can guess or predict the nonce value k used to generate a signature (r, s), the attacker can compute the signer's long-term secret key

$$x = r^{-1}(ks - H(m)) \bmod q. \tag{1}$$

Breitner and Heninger [BH19] found numerous Bitcoin signatures using 64-bit or smaller signature nonces, including several whose values appeared to be hand-generated.

Several million signatures in the Bitcoin blockchain use the signature nonce value $(q-1)/2$, where q is the order of the `secp256k1` curve. The x-coordinate of $(q-1)/2 \cdot G$ is 166 bits long, where one would expect a random point to have 256 bits, resulting in a much shorter signature than expected. This value is apparently used intentionally to collect small amounts of funds from addresses that will be then abandoned, and appears to be used because reducing the length of the digital signature reduces the transaction costs [blo15]. Many Bitcoin private keys are intended to be used only once, so the users of this nonce value appear to be using it intentionally because they do not care about compromising the secret key.

5.8 Repeated Nonces

If a victim ever signs two distinct message hashes $H(m_1)$ and $H(m_2)$ using the same (EC)DSA private key and signature nonce k to generate signatures (r_1, s_1) and (r_2, s_2), then it is trivial to recover the long-term private signing key from the messages. If the signatures have been generated using the same k value, then this is easy to recognise because $r_1 = r_2$. Then the value k can be recovered as

$$k = (H(m_1) - H(m_2))(s_1 - s_2)^{-1} \bmod q.$$

Once k has been recovered, then the secret key x can be recovered as in Eq. (1).

Repeated DSA and ECDSA signature nonces have been found numerous times in the wild.

In 2011, the ECDSA signature implementation used by the Sony PS3 was found to always use the same nonce k, revealing the code-signing key.

In 2012, the private keys of 1.6%, around 100 000, of SSH hosts with DSA host keys were compromised via repeated signature nonces [HDWH12]. Most of these vulnerable hosts appeared to be compromised due to poorly seeded deterministic pseudorandom-number generators. First, numerous hosts appear to have generated identical public keys using a pseudorandom-number generator seeded with the same value. Next, if the same poorly seeded pseudorandom-number generator is used for signature generation, then multiple distinct hosts will generate the same deterministic sequence of signature nonces. There were also multiple SSH implementations that were observed always using the same signature nonce for all signatures. In 2018, Breitner and Heninger [BH19] found 80 SSH host public keys were compromised by repeated signature nonces from ECDSA or DSA signatures, suggesting that the random-number generation issues found in 2012 may have been mitigated. They found a small number of repeated ECDSA signature nonces from HTTPS servers that compromised 7 distinct private keys used on 97 distinct IP addresses.

Repeated signature nonces in the Bitcoin blockchain have been well documented over the course of many years [BHH+14, CEV14, CV16, BR18], and empirical evidence suggests that attackers are regularly scanning the Bitcoin blockchain and immediately stealing Bitcoins from addresses whose private keys are revealed through repeated signature nonces. These vulnerabilities have been traced to at least two high-profile random-number generation vulnerabilities: a 2013 bug in the Android SecureRandom random-number generator [Kly13, MMS13] and a 2015 incident in which the Blockchain.info Android application had been attempting to seed from `random.org`, but was instead seeding from a 403 Redirect page to the HTTPS URL [Tea15].

To prevent this vulnerability, (EC)DSA implementations are recommended to avoid generating nonces using a random-number generator, and instead to derive signature nonces deterministically using a cryptographically-secure key derivation function applied to the message and the secret key [Por13].

5.9 Nonces with Shared Prefixes and Suffixes

If an implementation generates signature nonces where some bits of the nonce are known, predictable, or shared across multiple signatures from the same private key, then there are multiple algorithms that can recover the private key from these signatures. Let us specialise to the case where the signature nonce k has some most significant bits that are 0, so there is a bound $B < q$ such that $k < B$. For a signature (r_i, s_i) on message hash h_i, the signature satisfies the relation

$$k_i = s_i^{-1} r_i x - s_i^{-1} h_i \bmod q,$$

where the k_i is known to be small, the secret key x is unknown, and the other values are known.

Howgrave-Graham and Smart [HGS01] showed that one can solve for the secret key using a lattice attack due to Boneh and Venkatesan [BV96]. Generate the lattice basis

$$M = \begin{bmatrix} q & & & & & & \\ & q & & & & & \\ & & \ddots & & & & \\ & & & q & & & \\ s_1^{-1} r_1 & s_2^{-1} r_2 & \dots & s_m^{-1} r_m & B/q & & \\ s_1^{-1} h_1 & s_2^{-1} h_2 & \dots & s_m^{-1} h_m & & B \end{bmatrix}.$$

The vector $v_x = (k_1, k_2, \dots, k_m, Bx/q, B)$ is a short vector generated by the rows of M, and when $|v_x| < \det M^{1/\dim M}$, we hope to recover v_x using a lattice reduction algorithm like LLL [LLL82] or BKZ [Sch87, SE94].

In 2018, Breitner and Heninger found 302 Bitcoin keys that were compromised because they had generated signatures with nonces that were short, or shared a prefix or a suffix with many other nonces [BH19]. These were hypothesized to be due to implementation flaws that used the wrong length for nonces, or memory-bounds errors that caused the buffer used to store the nonce to overlap with other information in memory.

They also found three private keys used by SSH hosts whose signature nonces all shared the suffix `f27871c6`, which is one of the constant words used in the calculation of a SHA-2 hash, suggesting that the implementation generating these nonces was attempting to use a buggy implementation of SHA-2. In addition, there were keys used for both Bitcoin and SSH that used 160-bit nonces, suggesting that a 160-bit hash function like SHA-1 may have been used to generate the nonces.

5.10 Bleichenbacher Biased Nonces

Bleichenbacher developed an algorithm that uses Fourier analysis to recover the secret key in cases of even smaller bias than the lattice attack described above works for [Ble00]. The main idea is to define a function for signature samples (r_i, s_i) on message hash h_i

$$f(s_i^{-1} r_i) = e^{\frac{-2\pi i s_i^{-1} h_i}{q}}.$$

It can be shown that this function will have a large Fourier coefficient $\hat{f}(x)$ at the secret key x, and will be close to 0 everywhere else.

Bleichenbacher's algorithm then uses a technique for finding significant Fourier coefficients even when one cannot compute the full Fourier transform of the function to find the secret key x.

Bleichenbacher's algorithm was inspired by the observation that many DSA and ECDSA implementations will simply generate an n-bit random integer as the nonce for an n-bit group order, and do not apply rejection sampling if the nonce is larger than the group order. This means that the distribution of the nonce k over many signatures will be non-uniform. That is, if the group order $q = 2^n - t$ and an implementation naively generates an n-bit nonce k, then the values between 0 and t are twice as likely to occur than all other values.

The countermeasure against this attack is for implementations to use rejection sampling to sample a uniform distribution of nonces modulo q.

Most common elliptic curves have group orders that are very close to powers of two, so a practical attack is likely infeasible for these curves, but prime-field DSA group-generation procedures generally do not put such constraints on the subgroup order.

5.11 Countermeasures

The fact that a long-term (EC)DSA private key is compromised if that key is ever used to generate signatures by using a faulty or even very slightly biased random-number generator is a severe usability problem for the signature scheme, and has led to numerous problems in the real world. To protect against this vulnerability, implementations must always use 'deterministic' (EC)DSA to generate nonces. Fortunately, this countermeasure is becoming more popular, and more recent signature schemes such as Ed25519 build this nonce-generation procedure directly into the signature-generation scheme from the start.

6 Conclusion

The real-world implementation landscape for public-key cryptography contains numerous surprises, strange choices, catastrophic vulnerabilities, oversights and mathematical puzzles. Many of the measurement studies we toured evaluating real-world security have led

to improved standards, more secure libraries, and a slowly improving implementation landscape for cryptography in network protocols.

The fact that it has taken decades to iron out the implementation flaws for our oldest and most well-understood public-key cryptographic primitives such as RSA and Diffie–Hellman raises some questions about this process in the future: with new public-key standards on the horizon, are we doomed to repeat the past several decades of implementation chaos and catastrophic vulnerabilities?

Fortunately, the future seems a bit brighter than the complicated history of public-key cryptographic deployments might suggest. Although the mathematical structure of our future public-key cryptography standards will look rather different from the factoring and cyclic-group based cryptography we use now, many of the general classes of vulnerabilities are likely to remain the same: random-number generation issues, subtle biases and rounding errors, precomputation trade-offs, backdoored constants and substitution attacks, parameter negotiation, omitted validation checks and error-based side channels. With an improved understanding of protocol integration, a focus on real-world threat models, developer usability, and provably secure implementations, we have a chance to get these new schemes mostly right before they are deployed into the real world.

Acknowledgements

I am grateful to Shaanan Cohney, Matthew Green, Paul Kocher, Daniel Moghimi, Keegan Ryan and the anonymous reviewers for helpful suggestions, anecdotes and feedback, and to Joppe Bos and Martijn Stam for organizing and kitten-herding the editing of this wonderful book.

References

[ABD+15] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 5–17, Denver, CO, USA, October 12–16, 2015. ACM Press. 12, 13, 14, 16, 17, 19

[ABM+03] Adrian Antipa, Daniel R. L. Brown, Alfred Menezes, René Struik, and Scott A. Vanstone. Validation of elliptic curve public keys. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 211–223, Miami, FL, USA, January 6–8, 2003. Springer, Heidelberg, Germany. 23

[Adk11] Heather Adkins. *An update on attempted man-in-the-middle attacks*, 2011. https://security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html. 10

[AFK+07] Kazumaro Aoki, Jens Franke, Thorsten Kleinjung, Arjen K. Lenstra, and Dag Arne Osvik. A kilobit special number field sieve factorization. In Kaoru Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 1–12, Kuching, Malaysia, December 2–6, 2007. Springer, Heidelberg, Germany. 20

[AM09] Divesh Aggarwal and Ueli Maurer. Breaking RSA generically is equivalent to factoring. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 36–53, Cologne, Germany, April 26–30, 2009. Springer, Heidelberg, Germany. 2

[AMPS18] Martin R. Albrecht, Jake Massimo, Kenneth G. Paterson, and Juraj Somorovsky. Prime and prejudice: Primality testing under adversarial conditions. In Lie et al. [LMBW18], pages 281–298. 2

[APPVP15] Martin R. Albrecht, Davide Papini, Kenneth G. Paterson, and Ricardo Villanueva-Polanco. Factoring 512-bit RSA moduli for fun (and a profit of $ 9,000). https://martinralbrecht.files.wordpress.com/2015/03/freak-scan1.pdf, 2015. 4

[ASS+16] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: Breaking TLS using SSLv2. In Holz and Savage [HS16], pages 689–706. 8, 9

[Bar13] Razvan Barbulescu. *Algorithmes de logarithmes discrets dans les corps finis*. PhD thesis, Université de Lorraine, France, 2013. 14

[BBD+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press. 13

[BCC+13] Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren. Factoring RSA keys from certified smart cards: Coppersmith in the wild. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 341–360, Bengalore, India, December 1–5, 2013. Springer, Heidelberg, Germany. 7

[BCC+15] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: A white paper for the black hat http: //bada55.cr.yp.to. In Liqun Chen and Shin'ichiro Matsuo, editors, *Security Standardisation Research - Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*, volume 9497 of *LNCS*, pages 109–139. Springer, 2015. 22

[BCR+18] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, and Richard Davis. *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography*, 2018. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf. 16, 19

[BCR+19] Elaine Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, Richard Davis, and Scott Simon. *Recommendation for Pair-Wise Key Establishment Using Integer Factorization Cryptography*, 2019. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Br2.pdf. 4

[BD11] Elaine Barker and Quynh Dang. Nist sp 800-57 part 3 revision 1: Recommendation for key management–application-specific key management guidances. Technical report, Gaithersburg, MD, USA, 2011. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf. 3

[BDL+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012. 25

[Bel98] Mihir Bellare. PSS: Provably secure encoding method for digital signatures. *Submission to the IEEE P1363a: Provably Secure Signatures Working Group*, 1998. 7

[Ber06] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Yung et al. [YDKM06], pages 207–228. 23

[BH19] Joachim Breitner and Nadia Heninger. Biased nonce sense: Lattice attacks against weak ECDSA signatures in cryptocurrencies. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 3–20, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019. Springer, Heidelberg, Germany. 26, 27, 28

[BHH+14] Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 157–175, Christ Church, Barbados, March 3–7, 2014. Springer, Heidelberg, Germany. 27

[BKK+09] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. On the security of 1024-bit RSA and 160-bit elliptic curve cryptography. Cryptology ePrint Archive, Report 2009/389, 2009. https://eprint.iacr.org/2009/389. 3

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 1–12, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany. 7, 8

[Ble00] Daniel Bleichenbacher. On the generation of one-time keys in dl signature schemes. Presentation at IEEE P1363 working group meeting, 2000. 28

[blo15] The most repeated r value on the blockchain. https://bitcointalk.org/index.php?topic=1118704.0, 2015. 26

[Bon98] Dan Boneh. The decision Diffie-Hellman problem. In *Third Algorithmic Number Theory Symposium (ANTS)*, volume 1423 of *LNCS*. Springer, Heidelberg, Germany, 1998. Invited paper. 16

[BR95] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 92–111, Perugia, Italy, May 9–12, 1995. Springer, Heidelberg, Germany. 7

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures: How to sign with RSA and Rabin. In Maurer [Mau96], pages 399–416. 11

[BR11] Elaine B. Barker and Allen L. Roginsky. Nist sp 800-131a. transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. Technical report, Gaithersburg, MD, USA, 2011. https://doi.org/10.6028/NIST.SP.800-131A. 3, 13

[BR18] Michael Brengel and Christian Rossow. Identifying key leakage of bitcoin users. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 623–643, Cham, 2018. Springer International Publishing. 27

[BSY18] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of bleichenbacher's oracle threat (ROBOT). In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 817–849, Baltimore, MD, USA, August 15–17, 2018. USENIX Association. 9

[BV96] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 129–142, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany. 27

[BV98] Dan Boneh and Ramarathnam Venkatesan. Breaking RSA may not be equivalent to factoring. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 59–71, Espoo, Finland, May 31 – June 4, 1998. Springer, Heidelberg, Germany. 2

[Cas15] Ryan Castellucci. Cracking cryptocurrency brainwallets. https://rya.nc/cracking_cryptocurrency_brainwallets.pdf, 2015. 26

[CDL+00] Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Walter M. Lioen, Peter L. Montgomery, Brian Murphy, Herman te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, Paul C. Leyland, Joël Marchand, François Morain, Alec Muffett, Chris Putnam, Craig Putnam, and Paul Zimmermann. Factorization of a 512-bit RSA modulus. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 1–18, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany. 3

[CEV14] Nicolas T. Courtois, Pinar Emirdag, and Filippo Valsorda. Private key recovery combination attacks: On extreme fragility of popular bitcoin key management, wallet and cold storage solutions in presence of poor RNG events. Cryptology ePrint Archive, Report 2014/848, 2014. https://eprint.iacr.org/2014/848. 27

[CGH18] Shaanan N. Cohney, Matthew D. Green, and Nadia Heninger. Practical state recovery attacks against legacy RNG implementations. In Lie et al. [LMBW18], pages 265–280. 6

[Chr17] Chrome Platform Status. Chrome platform status: Remove dhe-based ciphers. https://www.chromestatus.com/feature/5128908798164992, 2017. 13, 21

[CKP+20] Shaanan Cohney, Andrew Kwong, Shahar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR_DRBG. In *2020 IEEE Symposium on Security and Privacy*, pages 1241–1258, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press. 6

[CMG+16] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual EC incident. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 468–479, Vienna, Austria, October 24–28, 2016. ACM Press. 6

[CN17] Jean-Sébastien Coron and Jesper Buus Nielsen, editors. *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany. 34, 35

[CNE+14] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In Fu and Jung [FJ14], pages 319–335. 6

[Com11] *Comodo Fraud Incident*, 2011. https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html. 10

[Cop97] Don Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology*, 10(4):233–260, September 1997. 4, 7

[CS06] An Commeine and Igor Semaev. An algorithm to solve the discrete logarithm problem with the number field sieve. In Yung et al. [YDKM06], pages 174–190. 14

[CV16] Ryan Castellucci and Filippo Valsorda. Stealing bitcoin with math. https://news.webamooz.com/wp-content/uploads/bot/offsecmag/151.pdf, 2016. 27

[CVE15] Cve-2015-3240. http://cve.mitre.org/cgi-bin/cvename.cgi?name=2015-3240, 2015. 18

[DA99] Tim Dierks and Christopher Allen. *RFC 2246 - The TLS Protocol Version 1.0*. Internet Activities Board, January 1999. 7, 11

[DAM+15] Zakir Durumeric, David Adrian, Ariana Mirian, James Kasten, Elie Bursztein, Nicolas Lidzborski, Kurt Thomas, Vijay Eranti, Michael Bailey, and J. Alex Halderman. Neither snow nor rain nor MITM...: An empirical analysis of email delivery security. In *Proceedings of the 2015 Internet Measurement Conference*, IMC, pages 27–39, New York, NY, USA, 2015. Association for Computing Machinery. 4

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. 11

[DL14] Antoine Delignat-Lavaud. *Mozilla Foundation Security Advisory 2014-73:RSA Signature Forgery in NSS*, 2014. https://www.mozilla.org/en-US/security/advisories/mfsa2014-73/. 10

[DR06] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.1. RFC 4346, RFC Editor, April 2006. https://www.rfc-editor.org/info/rfc4346. 1

[DR08] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, RFC Editor, August 2008. https://www.rfc-editor.org/info/rfc5246. 1, 8, 11, 12

[DWH13] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast internet-wide scanning and its security applications. In Samuel T. King, editor, *USENIX Security 2013*, pages 605–620, Washington, DC, USA, August 14–16, 2013. USENIX Association. 5

[Edg14] Jake Edge. A system call for random numbers: getrandom(). https://lwn.net/Articles/606141/, July 2014. 6

[Edl19] Bernd Edlinger. *Change DH parameters to generate the order q subgroup instead of 2q*, 2019. https://github.com/openssl/openssl/pull/9363. 16

[ElG84] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 10–18, Santa Barbara, CA, USA, August 19–23, 1984. Springer, Heidelberg, Germany. 12

[FGHT17] Joshua Fried, Pierrick Gaudry, Nadia Heninger, and Emmanuel Thomé. A kilobit hidden SNFS discrete logarithm computation. In Coron and Nielsen [CN17], pages 202–231. 20, 21

[Fin06] Hal Finney. *Bleichenbacher's RSA Signature Forgery Based on Implementation Error*, 2006. https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE. 4, 9

[FJ14] Kevin Fu and Jaeyeon Jung, editors. *USENIX Security 2014*, San Diego, CA, USA, August 20–22, 2014. USENIX Association. 33, 37

[FPS06] M. Friedl, N. Provos, and W. Simpson. Diffie-hellman group exchange for the secure shell (ssh) transport layer protocol. RFC 4419, RFC Editor, March 2006. https://www.rfc-editor.org/info/rfc4419. 12, 14, 20

[Fri20] Joshua Fried. *Personal communication*, 2020. 15

[GGK+16] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the lip of the volcano: Chosen ciphertext attacks on apple iMessage. In Holz and Savage [HS16], pages 655–672. 7

[Gil16] D. Gillmor. Negotiated finite field diffie-hellman ephemeral parameters for transport layer security (TLS). RFC 7919, RFC Editor, August 2016. https://www.rfc-editor.org/info/rfc7919. 10, 11, 12, 13, 14, 21

[Gor93] Daniel M. Gordon. Designing and detecting trapdoors for discrete log cryptosystems. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 66–75, Santa Barbara, CA, USA, August 16–20, 1993. Springer, Heidelberg, Germany. 21

[HC98] Dan Harkins and Dave Carrel. The Internet Key Exchange (IKE). IETF RFC 2409 (Proposed Standard), 1998. 9, 11, 12, 13

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security 2012*, pages 205–220, Bellevue, WA, USA, August 8–10, 2012. USENIX Association. 3, 5, 6, 7, 8, 24, 27

[Hef10] Craig Heffner. Littleblackbox: Database of private SSL/SSH keys for embedded devices. http://code.google.com/p/littleblackbox, 2010. 5

[HFH16] Marcella Hastings, Joshua Fried, and Nadia Heninger. Weak keys remain widespread in network devices. In *Proceedings of the 2016 Internet Measurement Conference*, IMC, pages 49–63, New York, NY, USA, 2016. Association for Computing Machinery. 6

[HG01] Nick Howgrave-Graham. Approximate integer common divisors. In Joseph H. Silverman, editor, *Cryptography and Lattices*, pages 51–66. Springer, Heidelberg, Germany, 2001. 7

[HGS01] N. A. Howgrave-Graham and N. P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, Aug 2001. 16, 27

[HS16] Thorsten Holz and Stefan Savage, editors. *USENIX Security 2016*, Austin, TX, USA, August 10–12, 2016. USENIX Association. 30, 34, 38, 39

[ICS20] ICSI. The icsi certificate notary. https://web.archive.org/web/20200624025519/https://notary.icsi.berkeley.edu/, 2020. 1, 22, 24

[Inf00] Information Technology Laboratory National Institute of Standards and Technology. *Digital Signature Standard (DSS)*, 2000. https://csrc.nist.gov/CSRC/media/Publications/fips/186/2/archive/2001-10-05/documents/fips186-2-change1.pdf. 21

[Inf13] Information Technology Laboratory National Institute of Standards and Technology. *Digital Signature Standard (DSS)*, 2013. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf. 2, 16, 21, 24, 25

[JKM18] Tibor Jager, Saqib A. Kakvi, and Alexander May. On the security of the PKCS#1 v1.5 signature scheme. In Lie et al. [LMBW18], pages 1195–1208. 9

[JSS15] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. Practical invalid curve attacks on TLS-ECDH. In Günther Pernul, Peter Y. A. Ryan, and Edgar R. Weippl, editors, *ESORICS 2015, Part I*, volume 9326 of *LNCS*, pages 407–425, Vienna, Austria, September 21–25, 2015. Springer, Heidelberg, Germany. 23

[KAF+10] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 333–350, Santa Barbara, CA, USA, August 15–19, 2010. Springer, Heidelberg, Germany. 12

[Kal98] B. Kaliski. PKCS #1: RSA encryption version 1.5. RFC 2313, RFC Editor, March 1998. https://www.rfc-editor.org/info/rfc2313. 7, 8, 9

[KDL+17] Thorsten Kleinjung, Claus Diem, Arjen K. Lenstra, Christine Priplata, and Colin Stahlke. Computation of a 768-bit prime field discrete logarithm. In Coron and Nielsen [CN17], pages 185–201. 12

[KHN+14] C. Kaufman, P. Hoffman, Y. Nir, P. Eronen, and T. Kivinen. Internet key exchange protocol version 2 (IKEv2). RFC 7296, RFC Editor, October 2014. https://www.rfc-editor.org/info/rfc7296. 11, 12, 13

[KKM11] Ann Hibner Koblitz, Neal Koblitz, and Alfred Menezes. Elliptic curve cryptography: The serpentine course of a paradigm shift. *Journal of Number Theory*, 131(5):781–814, 2011. 22

[Kle07] Thorsten Kleinjung. Discrete logarithms in gf(p) — 160 digits. https://listserv.nodak.edu/cgi-bin/wa.exe?A2=NMBRTHRY;1c737cf8.0702, 2007. 12

[Kli17] Evan Klitzke. Bitcoin transaction malleability. https://eklitzke.org/bitcoin-transaction-malleability, 2017. 25

[Kly13] Alex Klyubin. Some SecureRandom thoughts. https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html, August 2013. 27

[KM16] Neal Koblitz and Alfred Menezes. A riddle wrapped in an enigma. *IEEE Security & Privacy*, 14(6):34–42, 2016. 22

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987. 22

[Koc20] Paul Kocher. *Personal communication*, 2020. 3, 7

[Len87] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126(3):649–673, 1987. 17

[Len01] Arjen K. Lenstra. Unbelievable security. Matching AES security using public key systems (invited talk). In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 67–86, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany. 16

[LH91] Arjen K. Lenstra and Stuart Haber. Comment on proposed Digital Signature Standard, 1991. Letter to NIST regarding DSS, 1991. 21

[LHA+12] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 626–642, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany. 3, 4, 5, 6, 12, 17, 24

[Liv16] Carl D. Livitt. *Preliminary Expert Report of Carl D. Livitt*, 2016. https://medsec.com/stj_expert_witness_report.pdf. 4

[LK08] M Lepinski and S Kent. Additional diffie-hellman groups for use with ietf standards. IETF RFC 5114, 2008. 16, 18, 21

[LL96] Chae Hoon Lim and Pil Joong Lee. Generating efficient primes for discrete log cryptosystems. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.8261, 1996. 21

[LL97] Chae Hoon Lim and Pil Joong Lee. A key recovery attack on discrete log-based schemes using a prime order subgroup. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 249–263, Santa Barbara, CA, USA, August 17–21, 1997. Springer, Heidelberg, Germany. 19

[LLK13] B. Laurie, A. Langley, and E. Kasper. Certificate transparency. RFC 6962, RFC Editor, June 2013. https://www.rfc-editor.org/info/rfc6962. 10

[LLL82] Arjen K. Lenstra, Hendrik W. Lenstra, Jr., and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, December 1982. 27

[LM10] M. Lochter and J. Merkle. Elliptic curve cryptography (ecc) brainpool standard curves and curve generation. RFC 5639, RFC Editor, March 2010. https://www.rfc-editor.org/info/rfc5639. 22

[LMBW18] David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors. *ACM CCS 2018*, Toronto, ON, Canada, October 15–19, 2018. ACM Press. 30, 32, 35

[LV01] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, September 2001. 12, 13

[Man01] James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 230–238, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany. 9

[Mau94] Ueli M. Maurer. Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete algorithms. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 271–281, Santa Barbara, CA, USA, August 21–25, 1994. Springer, Heidelberg, Germany. 11

[Mau96] Ueli M. Maurer, editor. *EUROCRYPT'96*, volume 1070 of *LNCS*, Saragossa, Spain, May 12–16, 1996. Springer, Heidelberg, Germany. 31, 39

[McL15] Tim McLean. Critical vulnerabilities in JSON web token libraries. https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/, 2015. 10

[Mil76] Gary L Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976. 2

[Mil86] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426, Santa Barbara, CA, USA, August 18–22, 1986. Springer, Heidelberg, Germany. 22

[Mil20] Damien Miller. OpenSSH 8.3 released (and ssh-rsa deprecation notice). https://lwn.net/Articles/821544/, May 2020. 1

[Mir12] Ilya Mironov. *Factoring RSA Moduli. Part II.*, 2012. https://windowsontheory.org/2012/05/17/factoring-rsa-moduli-part-ii/. 2

[MKJR16] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. Pkcs #1: Rsa cryptography specifications version 2.2, November 2016. https://www.rfc-editor.org/info/rfc8017. 7

[MMS13] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. Randomly failed! The state of randomness in current java implementations. In Ed Dawson, editor, *CT-RSA 2013*, volume 7779 of *LNCS*, pages 129–144, San Francisco, CA, USA, February 25 – March 1, 2013. Springer, Heidelberg, Germany. 27

[Moz17] Mozilla. Http public key pinning (hpkp). https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning, 2017. 10

[MSW+14] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In Fu and Jung [FJ14], pages 733–748. 9

[MVVP12] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 62–72, Raleigh, NC, USA, October 16–18, 2012. ACM Press. 19

[Nat15] National Security Agency Central Security Service. Cryptography today. https://web.archive.org/web/20151123081120/https://www.nsa.gov/ia/programs/suiteb_cryptography, 2015. 22

[Nat20] National Security Agency. Patch critical cryptographic vulnerability in microsoft windows clients and servers. https://media.defense.gov/2020/Jan/14/2002234275/-1/-1/0/CSA-WINDOWS-10-CRYPT-LIB-20190114.PDF, 2020. 25

[NSS+17] Matús Nemec, Marek Sýs, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith's attack: Practical factorization of widely used RSA moduli. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1631–1648, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press. 3

[Ope20] *OpenSSH Legacy Options*, 2020. https://www.openssh.com/legacy.html. 13

[Ora14] Oracle. Jdk 8 security enhancements. https://docs.oracle.com/javase/8/docs/technotes/guides/security/enhancements-8.html, 2014. 13

[Orm98] H. Orman. *RFC 2412 - The Oakley Key Determination Protocol*. Internet Engineering Task Force, November 1998. http://www.ietf.org/rfc/rfc2412.txt. 13, 21

[PH78] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over gf(p) and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978. 15, 16

[Pol74] John M Pollard. Theorems on factorization and primality testing. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 76, pages 521–528. Cambridge University Press, Cambridge, UK, 1974. 2, 15

[Pol78] John M. Pollard. Monte Carlo Methods for Index Computation (mod p). *Mathematics of Computation*, 32:918–924, 1978. 14, 16

[Por13] T. Pornin. Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). RFC 6979, RFC Editor, August 2013. https://www.rfc-editor.org/info/rfc6979. 24, 25, 27

[Qua20] Qualys. Ssl pulse. https://www.ssllabs.com/ssl-pulse/, 2020. 4

[Rab80] Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980. 2

[Res15] Eric Rescorla. Nss accepts export-length dhe keys with regular dhe cipher suites ("logjam"). https://bugzilla.mozilla.org/show_bug.cgi?id=1138554, 2015. 13

[Res18] E. Rescorla. The transport layer security (TLS) protocol version 1.3. RFC 8446, RFC Editor, August 2018. https://www.rfc-editor.org/info/rfc8446. 11, 12

[RGB+16] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In Holz and Savage [HS16], pages 1–18. 16

[ric16] rico666. Large bitcoin collider. https://lbc.cryptoguru.org/, 2016. 26

[Rie16] Gerhard Rieger. *Socat security advisory 7 - Created new 2048bit DH modulus*, 2016. https://www.openwall.com/lists/oss-security/2016/02/01/4. 17

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, 1978. 2

[Sch87] Claus-Peter Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53:201–224, 1987. 27

[SDH16] Drew Springall, Zakir Durumeric, and J. Alex Halderman. Measuring the security harm of TLS crypto shortcuts. In *Proceedings of the 2016 Internet Measurement Conference*, IMC 2016, pages 33–47, New York, NY, USA, 2016. Association for Computing Machinery. 15

[SE94] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66:181–199, 1994. 27

[SEC00] SECG. SEC 2: Recommended elliptic curve domain parameters. *Standards for Efficient Cryptography Group, Certicom Corp*, 2000. 22, 25

[Sec15] Secure sockets layer (SSL)/transport layer security (TLS). http://www.spiegel.de/media/media-35511.pdf, 2015. 10

[Sho01] Victor Shoup. *A Proposal for an ISO Standard for Public Key Encryption (version 2.1)*,, 2001. https://www.shoup.net/papers/iso-2_1.pdf. 11

[SNS+16] Petr Svenda, Matús Nemec, Peter Sekan, Rudolf Kvasnovský, David Formánek, David Komárek, and Vashek Matyás. The million-key question - investigating the origins of RSA public keys. In Holz and Savage [HS16], pages 893–910. 3, 7

[tb13] thatch45 and basepi. *Change key generation seq*, 2013. https://github.com/saltstack/salt/commit/5dd304276ba5745ec21fc1e6686a0b28da29e6fc. 4

[Tea15] Blockchain Team. Android wallet security update. https://blog.blockchain.com/2015/05/28/android-wallet-security-update/, 2015. 27

[tra15] tranogatha. Establish deprecation date for DHE cipher suites in WebRTC. https://bugzilla.mozilla.org/show_bug.cgi?id=1227519, 2015. 13, 21

[VAS+17] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J. Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. In *NDSS 2017*, San Diego, CA, USA, February 26 – March 1, 2017. The Internet Society. 15, 16, 17, 18, 19, 22, 23

[VCL+16] Luke Valenta, Shaanan Cohney, Alex Liao, Joshua Fried, Satya Bodduluri, and Nadia Heninger. Factoring as a service. In Jens Grossklags and Bart Preneel, editors, *FC 2016*, volume 9603 of *LNCS*, pages 321–338, Christ Church, Barbados, February 22–26, 2016. Springer, Heidelberg, Germany. 3, 4

[VPN10] Intro to the VPN exploitation process. Media leak, September 2010. http://www.spiegel.de/media/media-35515.pdf. 10

[vW96] Paul C. van Oorschot and Michael J. Wiener. On Diffie-Hellman key agreement with short exponents. In Maurer [Mau96], pages 332–343. 16

[Wil14] Kathleen Wilson. Phasing out certificates with 1024-bit rsa keys. https://blog.mozilla.org/security/2014/09/08/phasing-out-certificates-with-1024-bit-rsa-keys/, 2014. 3

[WS96] David Wagner and Bruce Schneier. Analysis of the SSL 3.0 protocol. In Doug Tygar, editor, *Proceedings of the Second USENIX Workshop on Electronic Commerce*, volume 1. USENIX Association, Berkeley, CA, 1996. 10

[YDKM06] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors. *PKC 2006*, volume 3958 of *LNCS*, New York, NY, USA, April 24–26, 2006. Springer, Heidelberg, Germany. 31, 33

[YL06] T. Ylonen and C. Lonvick. The secure shell (ssh) transport layer protocol. RFC 4253, RFC Editor, January 2006. https://www.rfc-editor.org/info/rfc4253. 1, 11, 12, 13, 24

[YRS+09] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 debian openssl vulnerability. In Anja Feldmann and Laurent Mathy, editors, *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 15–27, New York, NY, USA, 2009. Association for Computing Machinery. 5, 6