New Bleichenbacher Records: Practical Fault Attacks on qDSA Signatures

Akira Takahashi[1], Mehdi Tibouchi[1,2] and Masayuki Abe[1,2]

[1] Kyoto University, Japan, takahashi.akira.58s@st.kyoto-u.ac.jp
[2] NTT Secure Platform Laboratories, Japan, {tibouchi.mehdi,abe.masayuki}@lab.ntt.co.jp

Abstract. In this paper, we optimize Bleichenbacher's statistical attack technique against (EC)DSA and other Schnorr-like signature schemes with biased or partially exposed nonces. Previous approaches to Bleichenbacher's attack suffered from very large memory consumption during the so-called "range reduction" phase. Using a carefully analyzed and highly parallelizable approach to this range reduction based on the Schroeppel–Shamir algorithm for knapsacks, we manage to overcome the memory barrier of previous work while maintaining a practical level of efficiency in terms of time complexity.

As a separate contribution, we present new fault attacks against the qDSA signature scheme of Renes and Smith (ASIACRYPT 2017) when instantiated over the Curve25519 Montgomery curve, and we validate some of them on the AVR microcontroller implementation of qDSA using actual fault experiments on the ChipWhisperer-Lite evaluation board. These fault attacks enable an adversary to generate signatures with 2 or 3 bits of the nonces known.

Combining our two contributions, we are able to achieve a full secret key recovery on qDSA by applying our version of Bleichenbacher's attack to these faulty signatures. Using a hybrid parallelization model relying on both shared and distributed memory, we achieve a very efficient implementation of our highly scalable range reduction algorithm. This allows us to complete Bleichenbacher's attack in the 252-bit prime order subgroup of Curve25519 within a reasonable time frame and using relatively modest computational resources both for 3-bit nonce exposure and for the much harder case of 2-bit nonce exposure. Both of these computations, and particularly the latter, set new records in the implementation of Bleichenbacher's attack.

Keywords: Digital Signature · Fault Attack · Bleichenbacher's Nonce Attack · Schroeppel–Shamir Algorithm · qDSA · Curve25519

1 Introduction

Attacks on Nonce Bias

Attacks on the nonces of (EC)DSA [Gal13] and other Schnorr-like signature schemes [Sch91] have been of interest to cryptanalysts over the last couple of decades. Since the knowledge of the nonces directly translates to the secret key, it is well known that the nonces should never be revealed or repeated. However, the nonces in Schnorr-like signatures are even more sensitive; in fact, it is possible to recover the secret key using only *partial* information of nonces. Perhaps the most famous example is the lattice attack initiated by Howgrave-Graham and Smart in [HGS01]. In a nutshell, the idea of lattice attacks is as follows. Given d Schnorr-like signatures of different messages with some least significant bits (LSB) of the nonces exposed, preprocess signature pairs to make the nonces *biased* in their the most significant bits (MSB) and construct the $(d+1)$-dimension lattice L such that *the hidden vector* $\boldsymbol{c}$ containing the information of the secret key belongs to L. If we can find the closest vector in L to the certain vector $\boldsymbol{v}$, which can be computed from the public information of the signature, it is heuristically likely to coincide with the hidden vector $\boldsymbol{c}$, which makes it possible to recover the secret key. Therefore, a key recovery of Schnorr-like signatures can be essentially reduced to solving the closest vector problem (CVP) in lattices. (See, e.g., [NT12] for more comprehensive description.) The lattice attack is a very powerful technique because it requires relatively few amount of signatures as input and works very efficiently in practice if many bits of the nonces are exposed. Since its first introduction, there have been quite a few works on the lattice attacks such as [NS02, NS03, NNTW05, BvdPSY14, BFMT16]. The instance of the largest group and the smallest nonce exposure broken by lattice attacks so far is the 160-bit DSA signature with 2-bit nonce exposure conducted by Liu and Nguyen [LN13]. However, if the number of exposed bits and the resulting bias is small, the lattice attacks are generally impractical due to the large lattice dimension, or because the hidden vector $\boldsymbol{c}$ does not necessarily coincide with the closest vector.

Aside from lattice attacks, Bleichenbacher presented a purely statistical attack technique against biased nonces in IEEE P1363 meeting in 2001 [Ble00]. This approach had never been published formally until recently revisited by [DMHMP14] and [AFG$^+$14]. The main idea of Bleichenbacher's attack is defining the "bias function" based on a Fourier notion of bias and searching for a candidate value of the secret key that displays the peak of the bias function. An advantage of Bleichenbacher's attack over lattice attacks is that it can in principle deal with arbitrary small biases and even work with non-uniformly biased inputs. On the negative side, Bleichenbacher's method requires many signatures as input, and therefore takes large space complexity due to its "range reduction" phase, where one has to find sufficiently many *small* and *sparse* linear combinations of signature values before computing the bias peak. For example, [AFG$^+$14] took a very straightforward approach to range reduction, which they call sort-and-difference, and successfully carried out a full key recovery of ECDSA over 160-bit curve using 1-bit bias. However, their approach required 2^{33} signatures as input and consumed 1TB of memory, which is a fairly unusual memory resource at the current time. Hence, Bleichenbacher's attack against groups of larger order and small biases, e.g., 256-bit curve and 2-bit bias, has been considered intractable.

Montgomery Curve, Curve25519, qDSA

Elliptic curve cryptography is widely deployed nowadays since it offers relatively short key length to achieve a good security strength. The most commonly-known instance is a signature scheme such as ECDSA. Most elliptic curve-based signature schemes operate based on a group law defined for a certain set of rational points on an elliptic curve over a finite field, and their security relies on the hardness of the elliptic curve discrete logarithm problem (ECDLP). Moreover, elliptic curves are used to achieve efficient key exchange protocols; for example, X25519 is specified in RFC7748 [LHT16] as a function that computes the scalar multiplication efficiently in the elliptic curve-based Diffie-Hellman key exchange (ECDHKE) [DH76]. The underlying curve used for X25519 is called Curve25519 [Ber06], which is one of the most famous instances of a Montgomery curve [Mon87]. Interestingly, Montgomery curves offer extremely fast scalar multiplication due to its *x-only arithmetic*; in fact, it does not involve y-coordinates at all and therefore requires no explicit group law, as opposed to the typical curve-based signature schemes. Although there have been several signature schemes that can operate over (twisted Edwards form of) Curve25519, such as EdDSA [BDL$^+$12], it was not until the quotient Digital Signature Algorithm (qDSA) [RS17] was proposed by Renes and Smith last year that one can reuse the public key of X25519-based ECDHKE for signatures without modifying the format at all. The qDSA is a high-speed and high-security signature scheme that relies on x-only arithmetic and can be instantiated with Montgomery curves (such as Curve25519) or Kummer surfaces. At a high-level, it closely resembles the Schnorr signature and is proved secure in the random oracle model as well. Due to the novelty, efficiency and compatibility with X25519, the qDSA is expected to be deployed in real-world constrained embedded systems, such as IoT devices. Some improvements to the signature generation and verification of qDSA have been recently proposed by [FFAL17].

Our Contributions

- Our first contribution is the optimized range reduction algorithm in Bleichenbacher's attack which overcomes the memory barrier of previous work while maintaining a practical level of efficiency in terms of time complexity. We designed the range reduction algorithm based on Howgrave-Graham–Joux's version [HGJ10] of Schroeppel–Shamir algorithm [SS81], which was originally proposed as a knapsack problem solver. The idea of making use of Schroeppel–Shamir was mentioned by Bleichenbacher himself, but it has never been concretely evaluated. Our approach has two merits. First, it takes less space complexity, and therefore requires fewer input signatures than the previous methods did in order to perform the same level of range reduction. Second, our algorithm can be parallelized in a very straightforward fashion with low communication overhead.

  We will first recall the Bleichenbacher's attack framework in Section 4, and then Section 5 describes our approach to the range reduction in detail and presents the theoretical results on the lower bound for the amount of input signatures for the algorithm to work correctly within Bleichenbacher's framework, including the performance comparison with previous nonce attack techniques.

- As a second contribution, we propose two concrete fault attack techniques against the qDSA instantiated with Curve25519 in order to induce 3-bit and 2-bit bias in its nonces. Our fault injection methods perturb the base point of Curve25519 to a point of non-prime order, so that its scalar multiplication by nonce reveals the few least significant bits (LSB) of it. The LSB obtained due to faults can be simply exploited to create bias in the most significant bits (MSB) of nonces. We will describe those two attacks in Section 3.

- Combining our two contributions, we are able to achieve a full secret key recovery on qDSA by applying our version of Bleichenbacher's attack to these faulty signatures. Using a hybrid parallelization model relying on both shared and distributed memory, we achieve a very efficient implementation of our highly scalable range reduction algorithm. This allows us to complete Bleichenbacher's attack in the 252-bit prime order subgroup of Curve25519 within a reasonable time frame and using relatively modest computational resources both for 3-bit nonce exposure and for the much harder case of 2-bit nonce exposure. To the best of the authors' knowledge, an attack against 252-bit group with such small exposures of the nonces has never been addressed before. Hence, both of these computations, and particularly the latter, set new records in the implementation of Bleichenbacher's attack.

Related Works

Bleichenbacher's nonce attack against DSA was first proposed in [Ble00]. De Mulder et al. revisited his method in [DMHMP14] and successfully performed a key recovery attack against ECDSA over NIST P-384 and brainpoolP384r1 using 4000 signatures with 5 bits of the nonces known. After that, Aranha et al. [AFG$^+$14] utilized Bleichenbacher's method to attack ECDSA over SECG P160 R1 with 2^{33} signatures of 1-bit biased nonces.

Recovering the secret key from the signatures knowing partial bits of the nonces reduces to an instance of hidden number problem (HNP) by Boneh and Venkatesan [BV96].

Howgrave-Graham and Smart first developed lattice attacks in [HGS01] to recover the DSA secret key over a 160-bit group using 30 signatures with 8 bits of the nonces known. Nguyen and Shparlinski in [NS02, NS03] later analyzed the lattice attacks in detail and presented the experimental results of the attack against 160-bit DSA using 100 signatures with only 3 bits of the nonces known. The instance of the largest group and the smallest nonce exposure broken (by lattice attacks) is the 160-bit DSA signature with 2-bit nonce exposure conducted by Liu and Nguyen [LN13]. Side-channel analysis and fault attacks have been often utilized in conjunction with lattice attacks to obtain the partial information of nonces. Such concrete attacks appear in, e.g., [NNTW05, BvdPSY14, BFMT16].

The first fault attack was discovered by Boneh, DeMillo and Lipton, which is often referred to as the Bellcore attack [BDL97]. This attack was against an implementation of RSA based on the Chinese Remainder Theorem. The various fault injection techniques and countermeasures are covered by [BCN+06]. In [FLRV08], Fouque et al. proposed the fault attack targeting the base point on non-twist-secure Montgomery curves. The idea of exploiting the low order points on Curve25519, upon which one of our fault attacks relies, was recently explored by Genkin, Valenta and Yarom [GVY17] in the context of attack against ECDH.

2 Preliminaries

2.1 Notations

In order to avoid confusion, we denote an index by an italic i and the imaginary unit by a roman i. A variant of big-O notation $\widetilde{O}$ will be used meaning that logarithm factors are omitted.

We denote b-LSB/MSB of an integer k by $\mathtt{LSB}_b(k)$ and $\mathtt{MSB}_b(k)$, respectively, assuming that k is represented as a fixed-length binary string. (The bit-length is typically 252-bit in this paper.) In Section 5, we will often use the binary representation of $(\tau+1)$-bit integer η, which is denoted as follows:

$$\begin{aligned}\eta &= \eta_\tau \| \dots \| \eta_1 \\ &= \sum_{i=1}^{\tau} \eta_i 2^{i-1}\end{aligned}$$

where $\eta_i \in \{0,1\}$ for $i = 1 \dots \tau$.

Moreover, we define a new notation $\eta_{[a:b]}$ to represent the substring of η and its corresponding value:

$$\begin{aligned}\eta_{[a:b]} &\coloneqq \eta_a \| \dots \| \eta_b \\ &= \sum_{i=b}^{a} \eta_i 2^{i-b}\end{aligned}$$

where $1 \le b \le a \le \tau$.

2.2 The quotient Digital Signature Algorithm

The quotient Digital Signature Algorithm (qDSA) is a variant of Schnorr signature scheme that operates on Kummer varieties and offers a key pair compatible with X25519-based Diffie–Hellman key exchange protocols [RS17]. We briefly recall the x-only arithmetic of Montgomery curves discovered in [Mon87] and the qDSA signature scheme instantiated with Curve25519 [Ber06], the most widely-known Montgomery curve. For more comprehensive introduction to Montgomery curves and Montgomery's ladder, see, e.g., [CS17] or [BL17].

Montgomery Curves and Their Arithmetic

Let p be a prime. A Montgomery curve over the finite field $\mathbb{F}_p$ is an elliptic curve defined by an affine equation

$$E_{A,B}/\mathbb{F}_p : By^2 = x^3 + Ax^2 + x$$

where the coefficient A and B are in $\mathbb{F}_p$ such that $A^2 \neq 4$ and $B \neq 0$.

Using the projective representation $(X : Y : Z)$, where $x = X/Z$ and $y = Y/Z$, we have the projective model

$$E_{A,B}/\mathbb{F}_p : BY^2Z = X^3 + AX^2Z + XZ^2$$

Note that the point at infinity $O = (0 : 1 : 0)$ is the only point where $Z = 0$.

Montgomery observed that the arithmetic in the above model does not involve y-coordinates. Namely, let $P = (X_P : Y_P : Z_P)$ and $Q = (X_Q : Y_Q : Z_Q)$ be two distinct points on $E_{A,B}$, the point addition and doubling are defined as follows:

$$\begin{aligned}
X_{P+Q} &= Z_{P-Q}[(X_P - Z_P)(X_Q + Z_Q) + (X_P + Z_P)(X_Q - Z_Q)]^2\\
Z_{P+Q} &= X_{P-Q}[(X_P - Z_P)(X_Q + Z_Q) - (X_P + Z_P)(X_Q - Z_Q)]^2\\
X_{[2]P} &= (X_P + Z_P)^2(X_P - Z_P)^2\\
Z_{[2]P} &= 4X_PZ_P((X_P - Z_P)^2 + ((A+2)/4)(4X_PZ_P))
\end{aligned}$$

where X_{P+Q}/Z_{P+Q}, X_{P-Q}/Z_{P-Q} and $X_{[2]P}/Z_{[2]P}$ are the x-coordinates of $P + Q$, $P - Q$ and $[2]P$, respectively.

Montgomery also proposed the algorithm, known as *Montgomery's ladder*, which efficiently computes the x-coordinate of the scalar multiplication $[k]P$ using only the point addition and doubling operations above. Therefore, it suffices to consider the points mapped into a one-dimensional projective space $\mathbb{P}^1(\mathbb{F}_p)$, which is simply the x-line. Formally speaking, let $E_{A,B}/\langle\pm1\rangle$ be the Kummer quotient of $E_{A,B}$ and $P = (X : Y : Z)$ an elliptic curve point, if the *quotient map* $\mathbf{x} : E_{A,B} \to E_{A,B}/\langle\pm1\rangle \cong \mathbb{P}^1(\mathbb{F}_p)$ is defined as

$$\mathbf{x} : P \mapsto \pm P = \begin{cases}(X : Z) & \text{if } P \neq O\\ (1 : 0) & \text{if } P = O\end{cases}$$

then the Montgomery's ladder efficiently computes the scalar multiplication on $\mathbb{P}^1$:

$$\texttt{Ladder} : (k, \pm P) \mapsto \pm[k]P$$

We omit the details of `Ladder` algorithm here. What readers should keep in mind is that it does not involve y-coordinates at all to compute the scalar multiplication.

qDSA Signature Generation

Algorithm 1 specifies the signature generation algorithm of qDSA. Here the domain parameters are

$$\mathcal{D} := (p, A, B, P, n, H)$$

where p is a large prime such that $\log_2 p \approx 252$, $A, B \in \mathbb{F}_p$ are coefficients that determine a Montgomery elliptic curve $E_{A,B}/\mathbb{F}_p$, $P \in E_{A,B}(\mathbb{F}_p)$ is a base point of prime order n, and $H : \{0,1\}^* \to \{0,1\}^{512}$ is a cryptographic hash function. The qDSA also uses the function `Compress` : $\mathbb{P}^1 \to \mathbb{F}_p$ to compress a projective point as follows:

$$\texttt{Compress} : \pm P = (X : Z) \mapsto x_P = XZ^{p-2}$$

Algorithm 1 qDSA signature generation

```
Input: $(d, d') \in \{0,1\}^{512}$: secret key, $x_Q$: compressed point of the public key $\pm Q = \pm[d]P$,
    $M \in \{0,1\}^*$: message to be signed, $\mathcal{D}$: domain parameters
Output: a valid signature $(x_R, s)$
1: $k \leftarrow H(d' || M) \mod n$
2: $\pm R = (X_R : Z_R) \leftarrow$ Ladder$(k, \pm P)$
3: $x_R \leftarrow$ Compress$(\pm R)$
4: $h \leftarrow H(x_R || x_Q || M)$                    ▷ ensure $\mathtt{LSB}_1(h) = 0$
5: $s \leftarrow k - hd \mod n$
6: return $(x_R, s)$
```

The value k at line 1 in Algorithm 1 is typically called *nonce*. From the line 5, the nonce obviously satisfies the following congruence relation:

$$k \equiv s + hd \mod n \tag{1}$$

Note that d' is only used as a seed and does not get involved in the verification at all. Hence, knowing d allows an attacker to generate a valid signature on arbitrary messages. In this paper, we will refer to d as the secret key for convenience.

Curve25519 Parameter Set

In the qDSA instance equipped with Curve25519, the parameters are specified as follows:

- $p = 2^{255} - 19$
- $(A, B) = (486662, 1)$
- $\pm P = (9 : 1)$
- Cofactor is 8, i.e., $\pm P$ has a prime order n, where n is slightly over 2^{252}
- $E_{A,B}(\mathbb{F}_p) \cong \mathbb{Z}/8\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$

2.3 Knapsack Problem

Although there exist several variants, we refer to *the computational 0–1 knapsack problem* as the knapsack problem. It can be stated as follows: Given a set of S positive integers $\{h_0, \ldots, h_{S-1}\}$ with some target value T, find the set of coefficients $\omega_i \in \{0,1\}$ such that the following holds:

$$T = \sum_{i=0}^{S-1} \omega_i h_i$$

3 Fault Attacks on qDSA

In this section, we describe several variants of a fault attack targeting the base point of scalar multiplication in qDSA signatures.

Our basic attack strategy is as follows. The qDSA signing algorithm uses the Montgomery ladder to compute the scalar multiplication $R = [k]P$ (up to sign), where k is the sensitive nonce value associated with the signed message; and the point R (or rather, its x-coordinate x_R) is output as part of the signature. Here, the correct base point P is a generator of the cyclic subgroup of order n in $E_{A,B}(\mathbb{F}_p) \cong \mathbb{Z}/8\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$.

Suppose that we can inject a fault into the device computing qDSA signatures so as to replace the point P by a different, faulty point $\widetilde{P}$ still on $E_{A,B}$, but with a different order, say $8n$. Then, even without knowing the exact value of $\widetilde{P}$, one can deduce information on k from the signature element x_R. For example, if x_R corresponds to a point of exact order n, we immediately get that k must be a multiple of 8: in other words, we obtain leakage information on the 3 least significant bits of k. As discussed in Section 3.3 below, such a bias can be turned into a bias on the *most* significant bits, which is enough to apply Bleichenbacher's attacks.

In the following sections, we describe several variants of this general approach, with a particular focus on how these attacks can be carried out in practice against practical implementations of qDSA. We also describe concrete fault attack experiments against a barely modified version of Renes and Smith's 8-bit AVR implementation of qDSA, on the XMEGA128D4 microcontroller of the ChipWhisperer-Lite low-cost side-channel and glitch attack evaluation board.

Before delving into those details, however, two preliminary remarks are in order.

First, we point out that our attack is rather novel in the sense that it relies on the new and unique structure of the qDSA signature scheme.

- On the one hand, the attack depends in a crucial way on the use of x-only arithmetic. Indeed, if we perturb a point P given by two coordinates, the resulting faulty point $\widetilde{P}$ will end up with overwhelming probability on a completely different curve among many possible choices, and even in a setting where the scalar multiplication by k still makes sense (as in the differential fault attack of Biehl et al. [BMM00]), the information on the curve on which $\widetilde{P}$ lies is lost in the signature, which contains only the x-coordinate of $\widetilde{R} = [k]\widetilde{P}$. This makes our strategy inapplicable to those settings.

- On the other hand, implementations using x-only arithmetic roughly divide into two families. Older, careless ones, tend to fall prey to the much simpler twist fault attack of Fouque et al. [FLRV08], in which case our strategy does apply, but is more complex and costly than necessary. Conversely, modern, careful implementations such as X25519 [Ber06] and other Diffie–Hellman implementations based on SafeCurves [BL], usually clear cofactors: in the description above, this means that the scalar k would be 8 times a uniformly random element of $\{0, \ldots, n-1\}$, and hence learning its 3 least significant bits would provide no information. That countermeasure thwarts our attack, even setting aside the fact that a few bits of leakage on Diffie–Hellman keys is much less of a security issue than nonce leakage in Schnorr-like signatures. Interestingly, the authors of qDSA apply that "clamping" technique to their secret keys [RS17, §3.3], but not to the nonces used in signature generation, which lets us carry out our attack.

Incidentally, the first point also explains why our attack applies to the genus 1 instantiation of qDSA (using Curve25519), but does not readily extend to the genus 2 instantiation (using the Gaudry–Schost Kummer surface). Indeed, the base point on the Kummer surface is represented by two coordinates, and injecting a fault will typically yield a point outside the surface, which prevents the attack for the same reason.

A second issue that should perhaps be stressed is that one can certainly consider much simpler fault attacks than our own on an unprotected implementation of qDSA: it is both easier and more effective to directly perturb the generation of the nonce k. For example, that generation typically ends with what essentially amounts to a copy of the final value into the array containing k (in the public qDSA implementations, this is done in the `group_scalar_get64` function). That array copy is a loop, and exiting the loop early results in a nonce with most of its bits equal to zero. It is then possible to recover the full secret key with as few as two signatures generated with those highly biased nonces, using e.g. the lattice attack of Howgrave-Graham and Smart [HGS01]. Note that this applies regardless of whether nonces are generated deterministically as in qDSA or probabilistically as in ECDSA.

However, the sensitivity of the nonce in Schnorr-like signature is very well-known, and one therefore expects a serious implementation that may be exposed to fault attacks to take appropriate countermeasures to protect against it (such as using double loop counters in the final array copy to check that the copy has completed successfully). On the contrary, our attack strategy is novel, and targets a part of the scalar multiplication that does not normally lead to serious attacks, as discussed above. It is thus much more likely to be left unprotected in real-world settings. Thus, we think that pointing out the corresponding threat is important, especially as qDSA is a scheme geared towards embedded devices (the target platforms of the accompanying implementations are AVR ATmega and ARM Cortex M0 microcontrollers [RS17, §7]).

3.1 Random Semi-Permanent Fault on the Base Point

Turning now to our attacks, we first describe a simple fault attack in a model that closely follows the strategy sketched at the beginning of this section.

Attack model. We suppose that the fault attacker is able to modify the base point P (represented by its x-coordinate on the quotient $E_{A,B}/\langle\pm 1\rangle \cong \mathbb{P}^1$) to a "somewhat random" faulty point $\widetilde{P}$, and then obtain several signatures computed with that faulty base point. We do *not* assume that the attacker knows the faulty point $\widetilde{P}$ once the fault is injected.

Realization of the model. Such a model can easily be realized in implementations where the representation of the base point is first loaded into memory (say at device startup) and then used directly whenever exponentiations are computed. This is a relatively common implementation pattern for embedded implementations of elliptic curve cryptography (for example, the `micro-ecc` library [Mac] works that way). It is then possible to induce a faulty base point either with faults on program flow at first load time (using e.g. clock or voltage glitches) so that some part of the corresponding array remains uninitialized/random, or with faults on memory (using e.g. optical attacks) so as to change some bit patterns within the existing array for P.

We note however that the model is more difficult to realize against the microcontroller implementations described in the original qDSA paper [RS17], due to the fact that the base point is recomputed before each signature generation. It may be possible to achieve a similar effect as above by carrying out a fault attack on *program memory*, so that e.g. the instruction that writes the byte `0x09` into the lowest-order byte of the base point is modified to write another byte instead (the same every time), but this presumably requires a significantly higher level of precision in the targeting of laser beams or x-rays.

Description of the fault attack. Suppose for simplicity that the fault attack yields a faulty base point $\widetilde{P}$ whose x-coordinate $\tilde{x}$ is uniformly random in $\mathbb{F}_p$ (we will see later on that the attack also works for values $\tilde{x}$ that aren't anywhere close to uniform).

In that case, we first observe that with probability close to $1/2$, $\tilde{x}$ is the abscissa of an actual point on the curve $E_{A,B}$, and it is otherwise the abscissa of a point on the quadratic twist of $E_{A,B}$. More precisely, excluding $\tilde{x} = 0$ (which corresponds to the point of order 2 both on the curve and its twist), the first case happens with probability exactly $(4n-2)/p$ and the second one with probability $(p+1-4n)/p$, both of which are in $[1/2 - 2^{-128}, 1/2 + 2^{-128}]$. From a signature generated with this faulty $\widetilde{P}$, it is easy to distinguish between the two cases, since we get the x-coordinate of $\widetilde{R} = [k]\widetilde{P}$, which will correspond to a point on $E_{A,B}$ when $\widetilde{P}$ itself is on the curve, and on the twist when $\widetilde{P}$ itself is on the twist.

If we get a point on the twist, we reject it by injecting another fault on the base point (restarting the device if necessary). We also reject faulty base points $\widetilde{P}$ that yield a value $\widetilde{R}$ of order at most 8 in the signature (in which case $\widetilde{P}$ itself must have been of order at most 8 since $k < n$); such exceptional points happen only with negligible probability anyway.

After this rejection, we know that $\widetilde{P}$ is on $E_{A,B}$, and has order $8n$, $4n$, $2n$ or n; its abscissa $\tilde{x}$ is uniformly distributed among the $4n - 4$ values in $\mathbb{F}_p$ corresponding to such points. Moreover, $2n - 2$ of these values correspond to points of exact order $8n$. Therefore, with probability $1/2$, $\widetilde{P}$ is of exact order $8n$, and again, it is easy to check that from generated signatures: simply compute $[4n](\pm\widetilde{R}) = \pm[4nk]\widetilde{P}$. If $\widetilde{P}$ is of order less than $8n$, this is always the point at infinity, whereas if it has order exactly n, this is the non-identity point of order 2 whenever $k < n$ is odd.

We can thus carry out another rejection step by generating e.g. 4 signatures with the faulty base point $\widetilde{P}$, and injecting another fault if for all of these signatures $[4n](\pm\widetilde{R})$ is the point at infinity. This always rejects points of order at most $4n$, and also rejects points of order $8n$ with probability $1 - 2^{-4}$.

Overall, after M fault injections on average, where:

$$M = \frac{p}{4n-5} \cdot 2 \cdot \frac{1}{1-2^{-4}} \approx 4.27 \tag{2}$$

we obtain a faulty base point $\widetilde{P}$ of order exactly $8n$.

Once such a point $\widetilde{P}$ is obtained, we claim that we can easily learn the 3 least significant bits of k for a constant fraction of the signatures generated with it.

Indeed, for each such signature, we can compute, up to sign, the point:

$$R' = [n](\pm\widetilde{R}) = \pm[nk]\widetilde{P},$$

which has order dividing 8. If it is the point at infinity or the point of exact order 2, both of which are equal to their inverses, we can directly obtain that $k \equiv 0 \pmod 8$ and $k \equiv 4 \pmod 8$ respectively. In other words, if R' is the point at infinity, we get $\mathtt{LSB}_3(k) = \mathtt{000}$, and if R' is the point of order 2, then $\mathtt{LSB}_3(k) = \mathtt{100}$. However, the points of exact order 4 and 8 are not invariant under $[\pm 1]$, so if R' is such a point, we cannot hope to learn 3 full bits of k; for example, if R' is of order 4, we only obtain $k \equiv 2$ or $6 \pmod 8$, but it is not possible to distinguish between both cases since we only get R' up to sign.

To obtain many signatures for which the 3 least significant bits of k are known, it then suffices to generate signatures with the faulty base point $\widetilde{P}$ and only keep those which satisfy that the point R' above is either the point at infinity or the point of order 2. This is the case whenever k is divisible by 4; thus, we keep a quarter of the generated signatures.

Once sufficiently many signatures have been collected, they can be used to carry out Bleichenbacher's attack as described in the following sections (see in particular Section 6.2 for concrete numbers of signatures, attack timings and memory consumption). A trivial but important point to note is that known LSBs by themselves do not translate into significant *bias* in the sense used in Bleichenbacher's attack (i.e. a large value for the bias function defined in Section 4.1). To achieve large bias, we first need to apply an affine transformation on signatures that map the partially known nonces k to values with their MSBs equal to zero (in this case, the 3 MSBs, since we have knowledge of 3 bits of k). This simple but crucial preprocessing step is described in Section 3.3 below.

Attack with a non-uniform faulty point. We have described the attack in the case when the fault injection yields a point $\widetilde{P}$ with uniformly random abscissa $\tilde{x}$ in $\mathbb{F}_p$. However, uniformity is far from crucial. The only important condition that should be satisfied is that the fault should result with significant probability in a point $\widetilde{P}$ of exact order $8n$.

Heuristically, this is expected to happen for essentially any "naturally occurring" subset of $\mathbb{F}_p$ of size much larger than 8. For example, consider the "fault on program memory" scenario alluded to above, in which the attacker is able to replace the correct base point of abscissa $x = 9$ by another base point $\widetilde{P}$ whose abscissa $\tilde{x}$ is a random integer still contained in a single byte (i.e. uniform between 0 and 255). The distribution is then very far from uniform in $\mathbb{F}_p$, but one can easily check that 119 such values $\tilde{x}$ correspond to a point on $E_{A,B}$ (and not its twist) with order at least n, and among them, 65 correspond to a point of order exactly $8n$. This means that the same attack can be carried out as above in that setting. The only change is the expected number of faults to inject, which instead of the estimate of Eq. 2 is slightly reduced to:

$$M = \frac{256}{65} \cdot \frac{1}{1 - 2^{-4}} \approx 4.20.$$

It is a bit difficult to justify the heuristic above in a rigorous way, but arithmetic techniques can be used to prove partial results in that direction. It follows from the character sum estimates of Kohel and Shparlinki [KS00] that if $\tilde{x}$ is picked uniformly at random in an interval of length $> p^{1/2+\varepsilon}$, then it corresponds to a point on $E_{A,B}$ of exact order $8n$ with probability $1/4 + O(p^{-\varepsilon})$. As a result, a fault attack inducing a value $\tilde{x}$ of that form works identically to the one where $\tilde{x}$ is uniform over $\mathbb{F}_p$, and the expected required number of faults is very close to the one given by Eq. 2.

3.2 Instruction Skipping Fault on Base Point Initialization

Although the fault model of the previous attack seems quite natural, it is difficult to realize against the implementations of qDSA described in the original paper [RS17], due to the fact that the representation of the base point P of Curve25519 is not stored in memory in a permanent way, but reconstructed every time a signature computation is carried out.

We now describe a fault attack that can easily be realized in practice on a very slightly modified version of the AVR ATmega implementation of qDSA distributed by the authors of the original paper. We also argue that the corresponding slight modification is plausible enough, and we mount the attack in practice on an XMEGA128 target using the ChipWhisperer-Lite low-cost side-channel and glitch attack evaluation board [OC14].

Attack model. The attack model is quite simple: the attacker injects a suitably synchronized fault upon signature generation that causes the reconstructed base point P to be incorrectly computed. The abscissa is set to $\tilde{x} = 1$ every time instead of the correct $x = 9$, and the signature is generated using the corresponding faulty base point $\widetilde{P}$. Note that this point $\widetilde{P}$ is of exact order 4.

In addition, we also assume that the attacker obtains a side-channel trace of the faulty execution of the signing algorithm. In that sense, the attack we will describe is a so-called *combined attack*, that uses both faults and side-channels. We note however that this isn't particularly restrictive: the synchronization of fault injection is typically carried out by waveform matching of side-channel traces anyway, so using the collected traced for additional purposes doesn't really strengthen the attack model.

Realization of the model. The entry point for the Montgomery ladder implementation used in the qDSA source code is the `ladder_base` function reproduced in Fig. 2a. Its main goal is to initialize the base point P and then call the ladder proper. More precisely, P is represented by its image in $E_{A,B}/\langle \pm 1 \rangle \cong \mathbb{P}^1$, with projective coordinates $(X : Z) = (9 : 1)$. To set P as such, the code first sets the X component (given by an array of 32 bytes) to 0 using the `fe25519_setzero` function, then the Z component to 1 with `fe25519_setone`, and finally modifies the least significant byte to 9.

```c
void ladder_base(
    ecp *r, const group_scalar *n
) {
    ecp base;
    fe25519 basex;

    fe25519_setzero(&base.X);
    fe25519_setone(&base.Z);
    base.X.v[0] = 9;
    fe25519_copy(&basex, &base.X);

    ladder(r, &base, &basex, n);
}
```

(a) Montgomery ladder entry point

```c
void ladder_base_modified(
    ecp *r, const group_scalar *n
) {
    ecp base;
    fe25519 basex;

    fe25519_setone(&base.X);
    fe25519_setone(&base.Z);
    base.X.v[0] = 9;
    fe25519_copy(&basex, &base.X);

    ladder(r, &base, &basex, n);
}
```

(b) Modified, functionally equivalent version

Figure 1: Initialization of the base point in qDSA's Montgomery ladder.

The idea of our attack is to uses glitches to skip the execution of that last step. On a platform like 8-bit AVR microcontrollers, this is relatively straightforward using clock glitches.

Doing so on the unmodified code of Fig. 2a results in a faulty base point $\widetilde{P}$ which maps to $(0 : 1)$ on $\mathbb{P}^1$ however: this is the point of exact order 2 on $E_{A,B}$, instead of a point of order 4 as desired. We can still obtain nonce leakage using that faulty base point, but only on a single bit of the nonce. That leakage isn't quite sufficient to deduce a practical attack.

Suppose however that the code was written as in Fig. 2b. The only change is that the X component of the base point is first set using `fe25519_setone` instead of `fe25519_setzero`. Of course, when executed correctly, the modified code is exactly functionally equivalent to the original one. However, skipping the instruction that changes the lowest order byte of X now results in a faulty base point $\widetilde{P}$ which maps to $(1 : 1)$ on $\mathbb{P}^1$: this is a point of order 4 as required.

That change might seem artificial, but there are plausible reasons why one might want to do it in practice. Most importantly, the function `fe25519_setzero` is almost never used elsewhere in the qDSA library code (there is exactly one other occurrence of it). Since reducing code size is a major concern for embedded implementations, removing that rarely used function and replacing its two uses by `fe25519_setone` (and adapting the code accordingly) makes sense. When compiling with `avr-gcc` 4.8.2, the change results in a code size reduction of 33 bytes, which can certainly justify such a change when program memory is at a premium.

Description of the combined attack. Since we are able to obtain signatures generated with the faulty base point $\widetilde{P}$ of order 4, the attack proceeds mostly as before. According to the description of qDSA, signatures will then contain $\pm\widetilde{R} = \pm[k]\widetilde{P}$, which is of order 4 when k is odd, of order 2 when $k \equiv 2 \pmod 4$, and the point at infinity when $k \equiv 0 \pmod 4$. In particular, we get $\mathtt{LSB}_2(k) = \mathtt{10}$ when $\pm\widetilde{R}$ is of order 2, and $\mathtt{LSB}_2(k) = \mathtt{00}$ when it is the point at infinity.

This should thus yield 2 LSBs of leakage on the nonce k whenever k is odd (i.e. for half of the generated signatures). After collecting sufficiently many such signatures and applying the affine transformation of Section 3.3 to obtain biased MSBs, we can then apply Bleichenbacher's attack. Concrete parameters, timings and memory consumption are provided in Section 6.1.

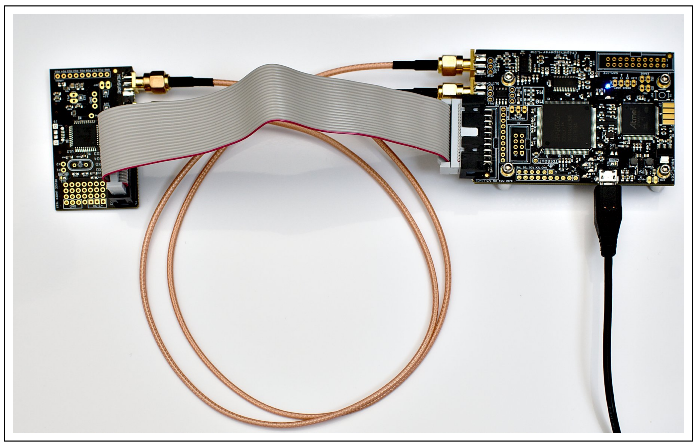


Figure 2: The ChipWhisperer-Lite evaluation board, connected to its XMEGA microcontroller target.

That simple description omits an important implementation detail that slightly complicates the attack, however. Namely, the point $\pm\widetilde{R} \in \mathbb{P}^1$ in signatures is represented in "affine coordinate" by a single element x_R of $\mathbb{F}_p$, and the point at infinity does not really have a well-defined representation in those terms. This is not an issue for correct executions of the qDSA algorithm, since the point at infinity happens with negligible probability; however, it is crucial in our specific attack setting. We therefore need to examine how x_R is computed from the projective representation $(X_R : Z_R)$ output by the Montgomery ladder.

In the qDSA implementation, x_R is computed by first inverting Z_R using Fermat's little theorem, and then multiplying the result by X_R. In other words, the code computes:

$$x_R \leftarrow \texttt{Compress}(X_R, Z_R) = Z_R^{p-2} \cdot X_R.$$

In the case of our faulty point $\pm\widetilde{R}$, we have:

$$(X_{\widetilde{R}}, Z_{\widetilde{R}}) = \begin{cases} (L_k, 0) & \text{when } k \equiv 0 \pmod 4 \\ (0, L_k) & \text{when } k \equiv 2 \pmod 4 \end{cases}$$

where in both cases $L_k \in \mathbb{F}_p$ is a large, typically full-size value depending only on k. In both cases, we therefore get:

$$x_{\widetilde{R}} = Z_{\widetilde{R}}^{p-2} \cdot X_{\widetilde{R}} = 0$$

and as a result, it is not possible to distinguish between the two cases just from the value included in the signature.

However, from an implementation perspective again, there is a clear difference between the two cases. When $k \equiv 0 \pmod 4$, the value $Z_{\widetilde{R}}$ for which the device computes the base field exponentiation $Z_{\widetilde{R}}^{p-2}$ is 0, whereas in the other case, it is a large, random-looking element L_k in $\mathbb{F}_p$. This difference should translate in a marked difference in power consumption and other side-channel emanations during the computation of this exponentiation operation!

Using side-channel leakage in addition to the fault, we are therefore able to distinguish between the two cases, and carry out the attack as expected.

Concrete glitch attack experiments. We successfully carried out the attack above on the implementation of qDSA for 8-bit AVR microcontroller platform [Ren17b], with the tweak of Fig. 2b. The cryptographic code was otherwise left entirely untouched, except for the insertion of a trigger in the signing algorithm (before the call to the `ladder_base` function) in order to facilitate the synchronization of injected faults. That synchronization should be doable directly in hardware from the acquired waveform (using e.g. oscilloscope SAD triggers) when using a more costly setup, but a manual software trigger comes in handy in our low-cost setting. Note that the qDSA implementation itself does not claim security against faults or physical attacks in general; however, conducting the attack on a real-world target allows us to confirm the validity of the fault model.

The attack was conducted on the ChipWhisperer-Lite side-channel and glitch attack evaluation board [OC14], which comes with an AVR XMEGA128D4 microcontroller target (Fig. 2). In order to use the accompanying software, we wrapped the qDSA code into a program running on the XMEGA target that can sign messages using the SimpleSerial serial console protocol supported by ChipWhisperer-Capture. The program supports several single character serial commands (followed by hexadecimal arguments), including in particular:

- `k`⟨32-byte hex string⟩: generate fresh key pair with the provided seed;
- `p`⟨16-byte hex string⟩: sign the provided message and return the first 32 bytes of the signatures (the rest of the 80-byte signature can be displayed with additional commands if necessary);
- `x`: reset the program;

and other miscellaneous commands for e.g. signature verification. One a key pair has been generated, a typical interaction on the serial console looks as follows (where the inputs are in blue and the outputs in black):

```
p8e230ea468bc5990f6a6820b5cb5f4b7
r614573B5BDB6E65F402BDBF2AFE3F67FCCD3F73B31680F16255EDF1B123B0658
z50
p3ae597975ad7c7574ee260cc14d724a1
rCAF7C938F5C180CB04E81586C2E6D0368D4CF0AB5C1A983BEA2FE1A0F2AA9C31
z50
```

where the blue lines ask for signatures on the provided messages, and the replies starting with `r` give with first 32 bytes of the computed signature (corresponding to the abscissa x_R). The lines starting with `z` signal the end of the response (and `50` indicates that the entire signatures are `0x50` = 80 bytes long).

We then use the glitch module of ChipWhisperer-Capture to generate clock glitches at selected positions during the execution of the program. After some trial and error, we find that XORed-in rectangular clock glitches of width 5% of the clock frequency, inserted at 2.5% of the corresponding clock cycles cause reliably reproducible misbehavior of the microcontroller. We then increment the position at which the glitch is inserted (as an offset from the trigger located right before the call to `ladder_base` in the signing algorithm), and observe the results on the serial console. At offset 202 clock cycles, we finally observe the required fault:

```
p3ae597975ad7c7574ee260cc14d724a1
r0000000000000000000000000000000000000000000000000000000000000000
z50
```

which we can confirm corresponds to skipping the assignment on step 9 of Fig. 2b. The power trace corresponding to the first few hundred cycles after the trigger is reproduced in Fig. 3, both for the correct execution and for the faulty one. One can clearly see a spike on the faulty trace when the glitch is injected, and how the skipped instruction results in

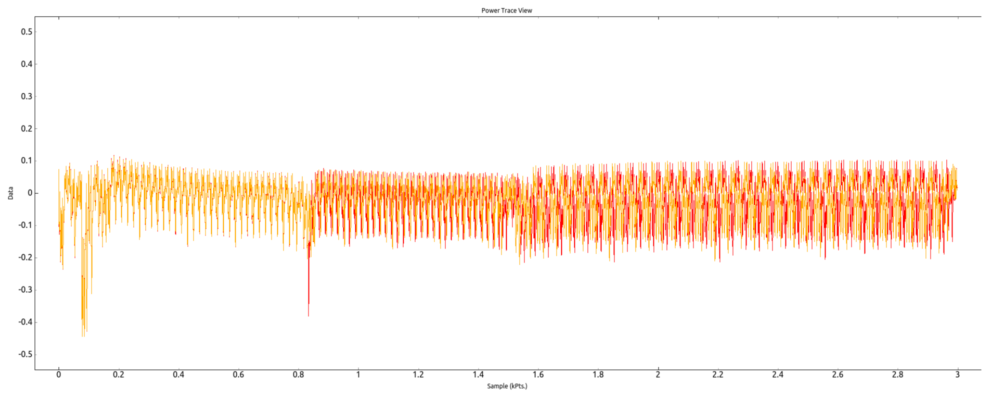


Figure 3: Power trace of the device starting from the call to `ladder_base`: correct execution (orange) and faulty one with glitch at offset 202 (red). Sampling rate is $4\times$ the clock frequency.

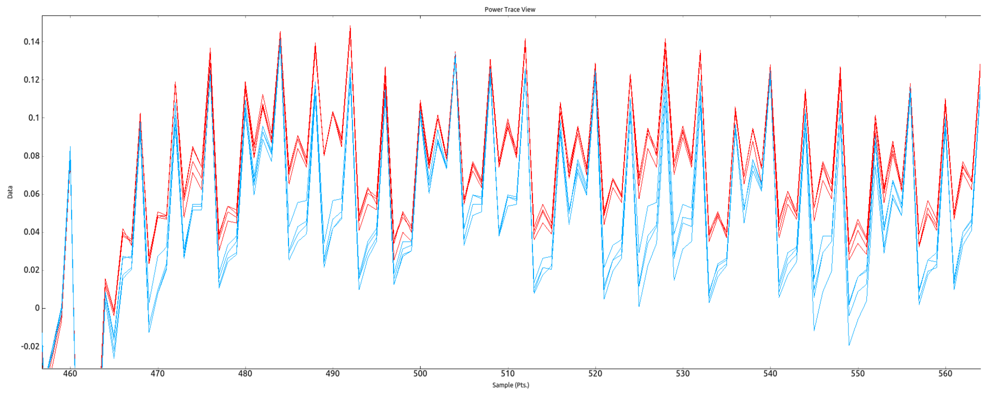


Figure 4: Power traces of the device around 130 cycles after the call to `compress`: blue (resp. red) traces correspond to $\widetilde{R}$ of order 2 (resp. at infinity).

a shift to the left of the trace of the faulty execution compared to the correct one after that point.

The fault is very reliably reproducible: in several hundred attempts at injecting the glitch, the assignment instruction was skipped 100% of the time, resulting in the same response

```
r0000000000000000000000000000000000000000000000000000000000000000
```

as expected.

To finish validating the combined attack, we then check that it is indeed easy to distinguish between the case when $\widetilde{R}$ is of order 2 on curve, and when it is the point at infinity. To do so, we plot corresponding power traces at a later point during the execution of the program, within the base field exponentiation used to compute the modular inverse of the coordinate $Z_{\widetilde{R}}$.

Fig. 4 shows two sets of several traces corresponding to faulty signatures of random messages. The traces in blue all correspond to the case when $\widetilde{R}$ is of order 2, and the traces in red to the case when it is the point at infinity. It is visually clear that the two sets of traces are easy to distinguish from each other, and that one can construct a very accurate distinguisher even from a small number of samples around that part of the execution.

3.3 Preprocessing Signatures for Bleichenbacher's Attack

Both of the attacks described above allow us to obtain multiple qDSA signatures for which a few LSBs of the nonces k are known. We would like to use those signatures with partial nonce exposure to retrieve the secret key d.

This problem can be seen as an instance of Boneh and Venkatesan's hidden number problem (HNP) [BV96]: given sufficiently many equations of the form (1), in which the pair (h, s) is known, and partial information on k is also given, recover d. Note that in our setting, the pair (h, s) is indeed known, since s is directly part of the signature, and h can be recomputed as $h = H(x_R \| x_Q \| M)$ (where x_R is again part of the signature; in particular, it is the faulty abscissa $x_{\widetilde{R}}$ in the case of faulty signatures).

The HNP algorithm used in this paper is essentially due to Bleichenbacher, and relies on a search for heavy Fourier coefficients. However, those heavy Fourier coefficients only reveal the secret key in an HNP instance where the *most* significant bits of nonces k are constant (say identically zero). Thus, our instance with known *LSBs* of k needs to be preprocessed in order to be amenable to Bleichenbacher's attack. This preprocessing stage is as follows.

Suppose that in our setting, the b least significant bits of nonces are known, i.e. $r := k \bmod 2^b$ is known for each nonce k. Subtracting r from Eq. (1) and dividing by 2^b, we get:

$$(k - r)/2^b \equiv (s - r)/2^b + hd/2^b \bmod n.$$

Now define $k' := (k - r)/2^b$, $s' := (s - r)/2^b$, and $h' := h/2^b$. The previous equation can be rewritten as:

$$k' \equiv s' + h'd \mod n$$

where $\mathtt{MSB}_b(k')$ is the all zero bit string. Hence, we get an equation of the correct form to apply Bleichenbacher's attack.

In the rest of this paper, we assume that S signatures are generated with either of the fault attacks, and preprocessed as above by the attacker. For simplicity, we change the notation and refer to $\{(h_i, s_i)\}_{i=0}^{S-1}$ as the set of *preprocessed* signatures, and $\{k_i\}_{i=0}^{S-1}$ as the biased nonces satisfying

$$\begin{cases} k_i & \equiv s_i + h_i d \mod n \\ \mathtt{MSB}_b(k_i) & = \mathtt{0}\ldots\mathtt{0} \end{cases} \quad \text{for} \quad 0 \le i \le S - 1.$$

3.4 Possible Countermeasures

Before turning to the description of Bleichenbacher's attack and of our optimizations thereof, we first mention a few countermeasures that can be applied to qDSA implementations in order to thwart the attacks of this section.

Since our attacks all target the base point in the Montgomery ladder computation, using generic techniques to protect that value should prevent the attack. Concrete ways of doing so include:

- carrying out consistency checks of proper execution when copying the value into memory (e.g. double loop counters);
- writing the value twice if it is reconstructed every time, so that a single instruction skip fault cannot corrupt it;
- computing a CRC checksum of the base point and checking that it gives the expected result before releasing a generated signature, etc.

Rather than these generic countermeasures, however, one could recommend instead to slightly modify the signing algorithm in a way that completely prevents attacks based on the existence of points of small order. Namely, instead of carrying out scalar multiplication by the nonce k, use $8k$ (or if using a curve E other than Curve25519, use $\alpha \cdot k$, where α is the least common multiple of the cofactors of E and its twist), and adjust the verification algorithm accordingly. This ensures that, even if the base point is tampered with somehow, the adversary will not be able to map the result of the scalar multiplication to a non-identity element of a subgroup of small order. This thwarts the attacks of this section in particular.

4 Bleichenbacher's Nonce Attack

In this part, we recall the Bleichenbacher's attack method. We also formulate the conditions required for the range reduction phase, which is by far the most costly phase in the attack. Note that Bleichenbacher's attack applies in principle to any Schnorr-like signatures with arbitrarily biased nonces, including (EC)DSA [Gal13], EdDSA [BDL+12], and ElGamal [ElG85], as long as they provide publicly available pairs (h, s) such that the congruence relation (1) holds.

Algorithm 2 specifies the high-level procedures of the attack. The step-by-step guide will be provided in the following subsections.

Algorithm 2 Bleichenbacher's nonce attack framework

```
Input:
    $\{(h_i, s_i)\}_{i=0}^{S-1}$ - the set of preprocessed Schnorr-like signatures with $b$-bit biased nonces
    $S$ - number of input signatures
    $L$ - number of linear combinations to be found
Output: $\ell$ most significant bits of $d$
 1: Range Reduction
 2: Find $L = 2^\ell$ reduced signatures $\{(h'_j, s'_j)\}_{j=0}^{L-1}$, where $(h'_j, s'_j) = (\sum_i \omega_{j,i} h_i, \sum_i \omega_{j,i} s_i)$
    is a pair of linear combinations with the coefficients $\omega_{j,i} \in \{0, \pm 1\}$, such that
    [C1] Small: $0 \le h'_j < L$
    [C2] Sparse: $|B_{n,b}(\boldsymbol{K})|^\Omega > 1/\sqrt{L}$, where $\Omega := \sum_i |\omega_{j,i}|$
    for $0 \le j \le L-1$

 3: Bias Computation
 4: $Z =: (Z_0, \ldots Z_{L-1}) \leftarrow (0, \ldots, 0)$
 5: for $j = 0$ to $L-1$ do
 6:     $Z_{h'_j} \leftarrow Z_{h'_j} + e^{2\pi \mathrm{i} s'_j / n}$
 7: end for
 8: $W \leftarrow \mathrm{iFFT}(Z) = (B_n(K_{w_0}), B_n(K_{w_1}), \ldots, B_n(K_{w_{L-1}}))$
 9: Find the value $m$ such that $|Z_m|$ is maximal
10: return $\mathtt{MSB}_\ell(\lfloor mn/L \rfloor)$
```

4.1 Bias Definition and Properties

We first formalize the bias of random variables in the form of discrete Fourier transform. Let us recall the definition of the bias presented at [Ble00] and its basic properties.

Definition 1. Let $\boldsymbol{X}$ be a random variable over $\mathbb{Z}/n\mathbb{Z}$. The bias $B_n(\boldsymbol{X})$ is defined as

$$B_n(\boldsymbol{X}) = E(e^{2\pi \mathrm{i} \boldsymbol{X}/n}) = B_n(\boldsymbol{X} \mod n)$$

where $E(\boldsymbol{X})$ represents the mean.

Likewise, the sampled bias of a set of points $V = \{v_i\}_{i=0}^{L-1}$ in $\mathbb{Z}/n\mathbb{Z}$ is defined by

$$B_n(V) = \frac{1}{L}\sum_{i=0}^{L-1} e^{2\pi \mathrm{i} v_i/n}$$

The bias as defined above satisfies the following properties. See [DMHMP14] for the proof.

Lemma 1. *Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be random variables.*

(a) If $\boldsymbol{X}$ is uniformly distributed over $\mathbb{Z}/n\mathbb{Z}$, then $B_n(\boldsymbol{X}) = 0$.

(b) If $\boldsymbol{X}$ and $\boldsymbol{Y}$ are independent, then $B_n(\boldsymbol{X}+\boldsymbol{Y}) = B_n(\boldsymbol{X})B_n(\boldsymbol{Y})$.

(c) $B_n(-\boldsymbol{X}) = \overline{B_n(\boldsymbol{X})}$, where $\overline{B_n(\boldsymbol{X})}$ denotes the complex conjugate of $B_n(\boldsymbol{X})$.

(d) Let $0 < T \leq n$ be a bound and $\boldsymbol{X}$ is uniformly distributed on the interval $[0, T-1]$. Then $|B_n(\boldsymbol{X})| = \frac{1}{T}\left|\frac{\sin(\pi T/n)}{\sin(\pi/n)}\right|$ and $|B_n(\boldsymbol{X})|$ is real-valued with $0 \leq |B_n(\boldsymbol{X})| \leq 1$.

The following claim is useful for approximating the bias value when the nonces are b-bit biased.

Corollary 1. *If n is sufficiently large and a random variable $\boldsymbol{K}$ corresponding to a nonce is uniformly distributed over $[0, n/2^b)$, the bias value $|B_{n,b}(\boldsymbol{K})|$ satisfies*

$$|B_{n,b}(\boldsymbol{K})| \approx \frac{2^b}{\pi}\sin(\pi/2^b)$$

Proof. Applying Lemma 1-(d),

$$\begin{aligned}|B_{n,b}(\boldsymbol{K})| &= \frac{2^b}{n}\left|\frac{\sin(\pi/2^b)}{\sin(\pi/n)}\right| \\ &\approx \frac{2^b}{\pi}\sin(\pi/2^b)\end{aligned}$$

In this paper, we focus on the case of $b = 2$ and $b = 3$. Then Corollary 1 gives us the bias values $|B_{n,2}(\boldsymbol{K})| \approx 0.9003$ and $|B_{n,3}(\boldsymbol{K})| \approx 0.9745$.

4.2 Range Reduction

The main idea of Bleichenbacher's attack is finding a secret key candidate that leads to the peak bias value: given a set of preprocessed pairs $\{(h_i, s_i)\}_{i=0}^{S-1}$ with biased nonces, we would like to find the candidate $w \in \mathbb{Z}/n\mathbb{Z}$ such that its corresponding set of nonce

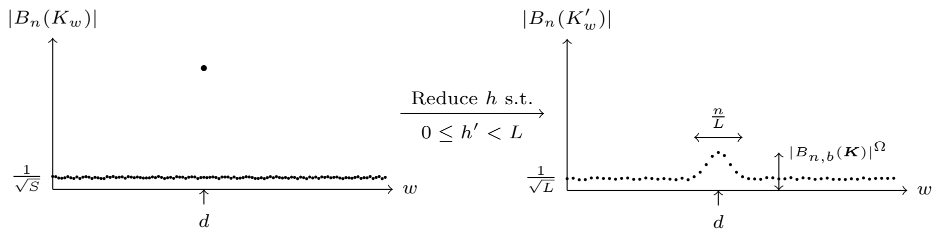


Figure 5: The effect of range reduction

candidates $K_w := \{s_i + h_i w\}_{i=0}^{S-1}$ shows a significant nonzero sampled bias. If w is equal to the true secret, i.e., $w = d$, we obtain a set of genuine biased nonces $K = \{k_i\}_{i=0}^{S-1}$ and its sampled bias $|B_n(K)|$ is close to 1, which we call the *peak*; if the guess is wrong, i.e., $w \neq d$, the sampled bias can be approximated by $1/\sqrt{S}$, which we call *noise*. Since Schnorr-like signatures allow anyone to compute a pair (h, s) that holds the relation (1), we thus have a way to determine the secret value d by evaluating $|B_n(K_w)|$ for all $w \in \mathbb{Z}/n\mathbb{Z}$ in a brute force way.

Condition 1: *Small* Linear Combinations However, checking all possible $w \in \mathbb{Z}/n\mathbb{Z}$ is computationally infeasible if n is large. Here a range reduction in Algorithm 2 plays an important role to avoid this problem. Bleichenbacher's observation is as follows: one can broaden the peak of the bias value by reducing the size of h values, so that it suffices to find a candidate *close* to d, instead of the exact solution. [DMHMP14] and [AFG$^+$14] examined his approach more concretely; they showed that by taking linear combinations modulo n of the original (h_i, s_i) pairs in a way that h'_j values are bounded by some L, as in the condition **[C1]**, the *width* of the peak gets broaden to approximately n/L, and therefore the peak area can be detected by evaluating the sampled bias of L-evenly-spaced values of w in $[0, n-1]$ [1]. Figure 5 illustrates this situation intuitively.

Condition 2: *Sparse* Linear Combinations Now let us look into the sparsity condition **[C2]**. Unfortunately, the range reduction has a negative side effect: the more dense the linear combinations become, the shorter the *height* of the peak gets. More concretely, **[C2]** can be shown as follows. Let us assume that $k_i = s_i + h_i d \mod n$ and a range reduction algorithm constructs the pair of linear combinations $(h'_j, s'_j) = (\sum_i \omega_{j,i} h_i, \sum_i \omega_{j,i} s_i)$, where $\omega_{j,i} \in \{0, \pm 1\}$ (we will omit the index j for simplicity). Then its corresponding nonce becomes $k' = \sum_i \omega_i k_i$. Let $\boldsymbol{K}_i$ be a uniformly distributed and independent random variable on $[0, n-1]$ that corresponds to k_i. Then applying (b) and (c) of Lemma 1, $B_n(\boldsymbol{K}_0) = B_n(\boldsymbol{K}_1) = \ldots = B_n(\boldsymbol{K}_{S-1})$ and

$$\begin{aligned} B_n\left(\sum_i \omega_i \boldsymbol{K}_i\right) &= B_n\left(\sum_{i^+} \boldsymbol{K}_{i^+} - \sum_{i^-} \boldsymbol{K}_{i^-}\right) \\ &= B_n\left(\sum_{i^+} \boldsymbol{K}_{i^+}\right) \cdot \overline{B_n\left(\sum_{i^-} \boldsymbol{K}_{i^-}\right)} \\ &= \prod_{i^+} B_n\left(\boldsymbol{K}_{i^+}\right) \cdot \overline{\prod_{i^-} B_n\left(\boldsymbol{K}_{i^-}\right)} \\ &= \prod_{i^+} B_n\left(\boldsymbol{K}_{i^+}\right) \cdot \prod_{i^-} \overline{B_n\left(\boldsymbol{K}_{i^-}\right)} \end{aligned}$$

where $i^+ \in \{i \mid \omega_i = 1\}$ and $i^- \in \{i \mid \omega_i = -1\}$. Hence taking the absolute value and assuming that the original nonces are b-bit biased, we obtain

$$\left| B_n\left(\sum_i \omega_i \boldsymbol{K}_i\right) \right| = |B_{n,b}(\boldsymbol{K})|^{\Omega}$$

, where $\Omega := \sum_i |\omega_i|$. This means that the height of the peak diminishes as the sum of coefficients Ω for the linear combination increases. Since the noise is approximately $1/\sqrt{L}$ and the peak value needs to serve as a distinguisher, we obtain the condition **[C2]** for the peak not to vanish.

[1] Rigorously speaking, [DMHMP14] only proved that the peak width gets broaden to $n/2L$, but [AFG$^+$14] empirically confirmed that checking L-evenly-spaced points is sufficient to detect the peak in practice.

In summary, finding small and sparse linear combinations for sufficiently small L (i.e., small enough for the FFT to be tractable) is the key to performing Bleichenbacher's attack efficiently. Let us briefly review the previous range reduction algorithms.

Sort-and-difference

We present the sort-and-difference algorithm conducted by [AFG+14] as the most straightforward instance of a range reduction algorithm. It simply works as shown below:

1. Sort the list[2] $\{(h_i, s_i)\}_{i=0}^{S-1}$ in ascending order by the h_i values
2. Take the successive differences to create a new list $\{(h'_j, s'_j)\}_{j=0}^{S-2} := \{(h_{i+1} - h_i, s_{i+1} - s_i)\}_{i=0}^{S-2}$
3. Repeat

With this approach, they successfully performed the key recovery attack against ECDSA on 160-bit curve with 1-bit nonce bias.

As a theoretical contribution, they analytically proved that approximately $(1 - e^{-2^\gamma})S$ signatures are obtained such that $h'_j < 2^{\log n - \log S + \gamma}$ after the first application of sort-and-difference, where $\gamma \in \mathbb{Z}$ is a parameter. However, because the h' values are not uniformly random and independently distributed anymore, their experimental result showed that the ratio $(1 - e^{-2^\gamma})$ does not hold after the second iterations and the actual ratio drops as the algorithm iterates, i.e., the number of reduced signatures such that $h'_j < 2^{\log n - \iota(\log S - \gamma)}$ after ι rounds is less than $(1 - e^{-2^\gamma})^\iota S$.

As a consequence, the sort-and-difference required $S = 2^{33}$ input signatures to satisfy **[C1]** and **[C2]** for their attack setting. Their implementation consumed nearly 1TB of RAM, and therefore attacking groups of larger order with small nonce biases was thought to be out of reach due to its huge memory consumption.

Lattice Reduction

De Mulder et al. in [DMHMP14] proposed to use lattice reduction to carry out the range reduction. They used the BKZ algorithm applied in lattices of dimension around 500 to mount Bleichenbacher's attack against 384-bit ECDSA with 5-bit nonce bias, using a total of about 4000 signatures as input.

The idea of using lattice reduction for range reduction may seem quite natural indeed: after all, range reduction is about finding very short and sparse linear combinations from a large list $\{h_i\}_{i=0}^{S-1}$ of integers, which seems closely related to the problem of finding very short vectors in the lattice generated by the rows of the following matrix:

$$\begin{bmatrix} \kappa & & 0 & h_0 \\ & \ddots & & \vdots \\ 0 & & \kappa & h_{S-1} \end{bmatrix}$$

for a suitable scaling constant κ. Indeed, any vector in that lattice is of the form $(\kappa\omega_0, \ldots, \kappa\omega_{S-1}, \sum_i \omega_i h_i)$, and it is thus short when all the ω_i's have a small absolute value and the linear combination $\sum_i \omega_i h_i$ is also short.

However, two problems arise when trying to apply that approach to more demanding parameters than the ones considered by De Mulder et al., particularly when the bias is significantly smaller.

First, the conditions above do not really capture the *sparsity* of the linear combinations, which is of paramount importance for small biases, since the bias function decreases

[2] We will often refer to an ordered set as a *list*

exponentially with the number of non zero coefficients. To get acceptably sparse linear combinations, one is led to start with a lattice of small dimension, constructed from a random subset of the h_i's of size at most equal to the desired weight of the linear combination. This in turns makes short vectors in that lattice no longer very short.

Second, although the coefficient ω_i's tend to be relatively small, they are not constrained to lie in $\{-1, 0, 1\}$ as in the previous description, and as a result it is no longer true that the bias of linear combinations is given by $|B_{n,b}(\boldsymbol{K})|^{\Omega}$, $\Omega = \sum |\omega_i|$, when the original nonces have b-bit bias. In fact, the bias can be computed explicitly, and it is smaller than this value in general. In particular, if one of the ω_i's is a multiple of 2^b, it is easy to check that the bias becomes exponentially small. Since for small b it is not usually feasible to avoid the appearance of such a coefficient, the linear combinations given by lattice reduction are typically not useful.

4.3 Bias Computation

Now let $w_m = mn/L$, with $m \in [0, L-1]$, be an L-evenly-spaced secrete key candidate in $[0, n-1]$ and $K_{w_m} := \{s'_j + h'_j w_m\}_{j=0}^{L-1}$ be a set of candidate nonces. Assuming that L reduced signatures have been obtained by a range reduction phase, the sampled bias is

$$\begin{aligned} B_n(K_{w_m}) &= \frac{1}{L} \sum_{j=0}^{L-1} e^{2\pi \mathrm{i}(s'_j + h'_j w_m)/n} \\ &= \sum_{t=0}^{L-1} \underbrace{\left(\frac{1}{L} \sum_{\{j \mid h'_j = t\}} e^{2\pi \mathrm{i} s'_j/n} \right)}_{Z_t} e^{2\pi \mathrm{i} tm/L} \end{aligned}$$

Thus, by constructing the vector $Z =: (Z_0, \ldots, Z_{L-1})$, the sampled biases $B_n(K_{w_m})$ for $m \in [0, L-1]$ can be computed all at once using the inverse Fast Fourier Transform (iFFT). Note that (i)FFT only takes $\widetilde{O}(L)$ time and $O(L)$ space complexities. Finally, recalling that the peak width is now broadened to n/L via range reduction, the algorithm picks the candidate w_m that leads to the largest sampled bias, so we can expect that w_m shares its ℓ-MSB with the secret d.

4.4 Recovering Remaining Bits

Once ℓ-MSB of the secret was recovered, i.e.we know $d_{[\lambda:\lambda-\ell+1]}$ such that $d = d_{[\lambda:\lambda-\ell+1]} 2^{\lambda-\ell} + d_{[\lambda-\ell:1]}$ (see Section 2.1 for the notation), the congruence relation (1) can be rewritten as follows:

$$\begin{aligned} k_i &\equiv s_i + h_i d \mod n \\ &\equiv s_i + h_i \left(d_{[\lambda:\lambda-\ell+1]} 2^{\lambda-\ell} + d_{[\lambda-\ell:1]} \right) \mod n \\ &\equiv \left(s_i + h_i d_{[\lambda:\lambda-\ell+1]} 2^{\lambda-\ell} \right) + h_i d_{[\lambda-\ell:1]} \mod n \end{aligned}$$

Hence, defining $s_i := \left(s_i + h_i d_{[\lambda:\lambda-\ell+1]} 2^{\lambda-\ell} \right)$, Algorithm 2 can proceed with the attack to recover the ℓ-MSB of $d_{[\lambda-\ell:1]}$, except that this time the FFT table is constructed in the following way: let $n' := 2^{\lambda-\ell}$ be the upper bound of $d_{[\lambda-\ell:1]}$ and $w'_m = mn'/L$ be a

L-evenly-spaced candidate in $[0, n'-1]$, then the sampled bias is

$$\begin{aligned} B_n(K_{w'_m}) &= \frac{1}{L}\sum_{j=0}^{L-1} e^{2\pi \mathrm{i}(s'_j + h'_j w'_m)/n} \\ &= \sum_{t=0}^{L-1} \underbrace{\left(\frac{1}{L} \sum_{\{j \mid \lfloor h'_j n'/n \rfloor = t\}} e^{2\pi \mathrm{i} s'_j/n} \right)}_{Z_t} e^{2\pi \mathrm{i} t m/L} \end{aligned}$$

As such, we only need to reduce the h values so that $0 \le h'_j < Ln/n' = L^2$, which should be much faster than the first round. By repeating the above operations, we can iteratively recover the ℓ-bit of the secret key d per each round.

5 Optimization and Parallelization of Bleichenbacher's Attack

As we discussed in the previous section, the range reduction is the most costly phase in Bleichenbacher's attack framework and the previous approaches to it are basically memory-bound. In this section, we present our approach to range reduction to overcome this memory barrier while maintaining a practical level of efficiency in terms of time complexity.

5.1 Our Approach: Using Schroeppel–Shamir Algorithm

We begin with an intuitive discussion on the nature of the problem of finding small and sparse linear combinations (we call it *the range reduction problem* for convenience). Interestingly, Bleichenbacher mentioned in [Ble00] the use of Schroeppel–Shamir algorithm, which was originally proposed as a knapsack problem solver in [SS81], would save memory in the range reduction phase, though there has been no concrete evaluation made on it until today. Let us develop his idea more concretely. The range reduction problem can be indeed regarded as a variant of the knapsack problem (as defined in Section 2.3) in a broad sense; instead of searching for the exact knapsack solutions, we would like to find sufficiently many *sparse* patterns of coefficients that lead to the linear combination *smaller* than a certain threshold value. With this in mind, we can transform Schroeppel–Shamir's knapsack problem solver into a range reduction algorithm. However, applying the original Schroeppel-Shamir algorithm introduces large priority queues (or min-heaps) to store partial linear combinations, which are not cache-friendly and moreover make it hard to optimize and parallelize the algorithm in practice. Hence, our approach is specifically inspired by the optimized version due to Howgrave-Graham and Joux, which replaced the priority queues with simple lists. Though their algorithm is intended for solving the knapsack problem, we observe that it happens to have two desirable characteristics in the context of Bleichenbacher's attack: modest space complexity and compatibility with large-scale parallelization. The interested reader is invited to refer to [HGJ10, §3] to become familiar with their approach in knapsack-specific setting. Figure 6 and Figure 7 depict how Schroeppel–Shamir algorithm and its variant by Howgrave-Graham–Joux would serve as a range reduction at a high level.

In a nutshell, the range reduction transformed from Howgrave-Graham–Joux's algorithm works as follows:

1. Split a set of $S = 2^{\alpha+2}$ input signatures into 4 lists $\mathcal{L}^{(1)}, \mathcal{R}^{(1)}, \mathcal{L}^{(2)}$, and $\mathcal{R}^{(2)}$ of size $S/4 = 2^{\alpha}$,

2. Create the lists $\mathcal{A}^{(r)}$, for each $r \in \{1, 2\}$, that consists of linear combinations of two $(\eta^{(r)}, \zeta^{(r)}) = \mathcal{L}^{(r)}[i] + \mathcal{R}^{(r)}[j] = (h_i^{(r)} + h_j^{(r)}, s_i^{(r)} + s_j^{(r)})$ such that $\eta^{(r)}$'s top consecutive $(\alpha + 1)$ bits coincide with a certain value $c \mod 2^\alpha$, and

3. Sort $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$ and search for the short differences between elements from them such that they are β-bit smaller than the original h values, where β is a parameter.

That is, it first collects the linear combinations of two to make sure that the collision happens in the top consecutive bits when taking differences, so that the resulting linear combinations of four are expected to be much smaller with good probability. We give the concrete procedures of our range reduction in Algorithm 3. It calls Algorithm 4 inside as the subroutine that collects the linear combinations of two such that their top consecutive $(\alpha + 1)$ bits coincide with a given value.

5.2 Analysis

We first show how to choose the appropriate parameter β so that the resulting number of reduced signatures approximately remains S and the space usage is stable in each round. We also evaluate the space and time complexity of Algorithm 3.

Theorem 1. *After the first round of Algorithm 3, the expected cardinality of* `sols`*, which we denote by* L*, is* $\frac{4}{3}\left(2^{4\alpha-\beta} + 2^{5\alpha-2\beta}\right) + \frac{2}{3}\left(2^{2\alpha-\beta} + 2^{3\alpha-2\beta}\right)$.

Proof. We first show that the expected cardinality of $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$, which are the lists of linear combinations of two, is $C = 2^\alpha$ after the line 14. Second, we evaluate the probability that a $(\tau - \beta)$-bit-bounded linear combination of four, which is composed of items in $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$, can be found, i.e., $\Pr\left[h' < 2^{\tau-\beta}\right]$.

When $\rho = 1$, $\tau = \lambda$ is the bit-length upper bound of h values. Since h values are outputs of a cryptographic hash function, they are uniformly distributed in $[0, 2^\tau)$ [3]. Hence, the values corresponding to the top α-bits of them, i.e., $\lfloor h/2^{\tau-\alpha} \rfloor$, are uniformly distributed in $[0, C)$.

Let $\eta = h_i + h_j$ be an integer represented as $(\tau + 1)$-bit string. Then the value corresponding to its top $(\alpha + 1)$-bits is $\lfloor \eta/2^{\tau-\alpha} \rfloor = \eta_{[\tau+1:\tau-\alpha+1]}$ (see Section 2.1 for the definition of the notation). Recalling that the sum of two uniform distributions follows a triangular distribution,

$$\begin{aligned}\Pr\left[\lfloor \eta/2^{\tau-\alpha} \rfloor = c\right] &= \Pr\left[\eta_{[\tau+1:\tau-\alpha+1]} = c\right] \\ &= \begin{cases} (c+1)/C^2 & \text{if } 0 \le c \le C-1 \\ 2/C - (c+1)/C^2 & \text{if } C \le c \le 2C-1 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

We can make the distribution above uniform by considering the modulo C, i.e., by ignoring $\eta_{\tau+1}$:

$$\begin{aligned}\Pr\left[\lfloor \eta/2^{\tau-\alpha} \rfloor \equiv c \mod C\right] &= \Pr\left[\eta_{[\tau:\tau-\alpha+1]} = c\right] \\ &= \begin{cases} 1/C & \text{if } 0 \le c \le C-1 \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

This corresponds to calling Algorithm 4 twice on c and $c + C$. There are $L' \times L'$ possible linear combinations of two between $\mathcal{L}$ and $\mathcal{R}$. Since $L' = L/4 = S/4 = C$ when $\rho = 1$, for

[3] We remark that this is not actually the case for qDSA since it ensures h values to be even, unlike typical Schnorr-like signatures. However, it is possible to make them uniformly distributed over a narrower range by using the filtering technique discussed in Section 5.5.

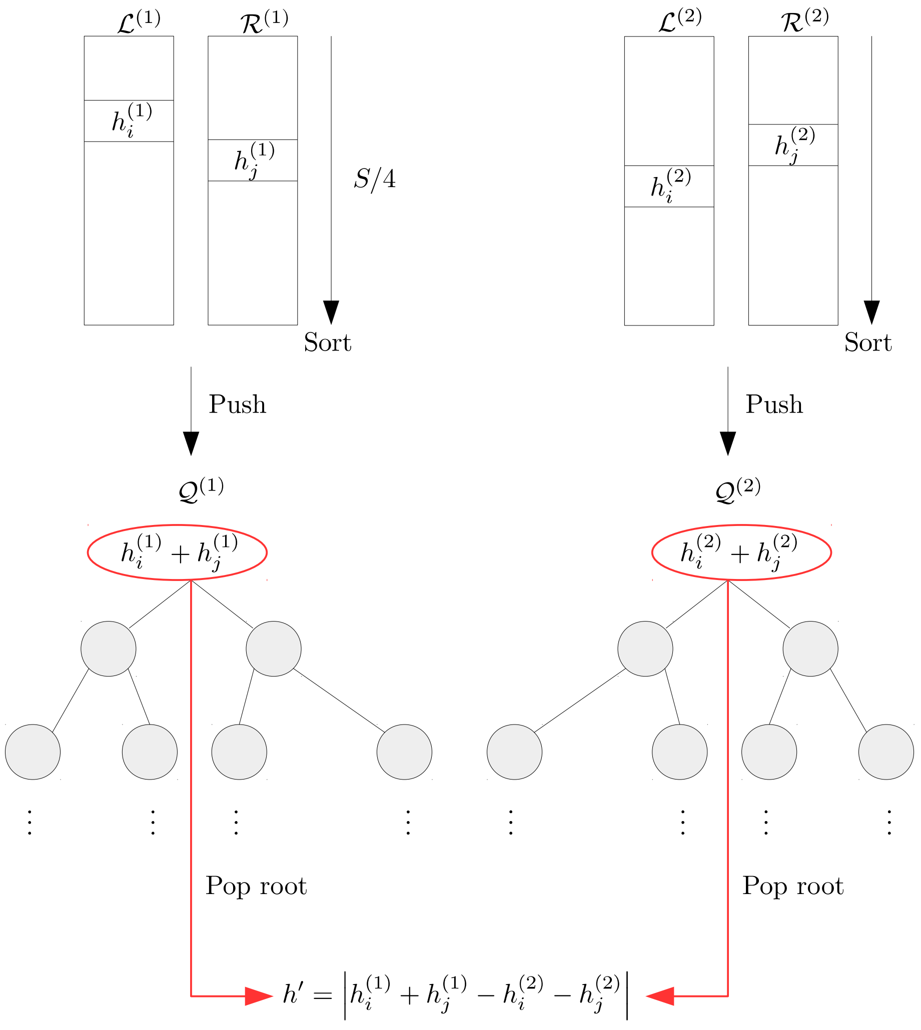


Figure 6: Overview of the Schroeppel–Shamir-based range reduction algorithm directly transformed from their original version

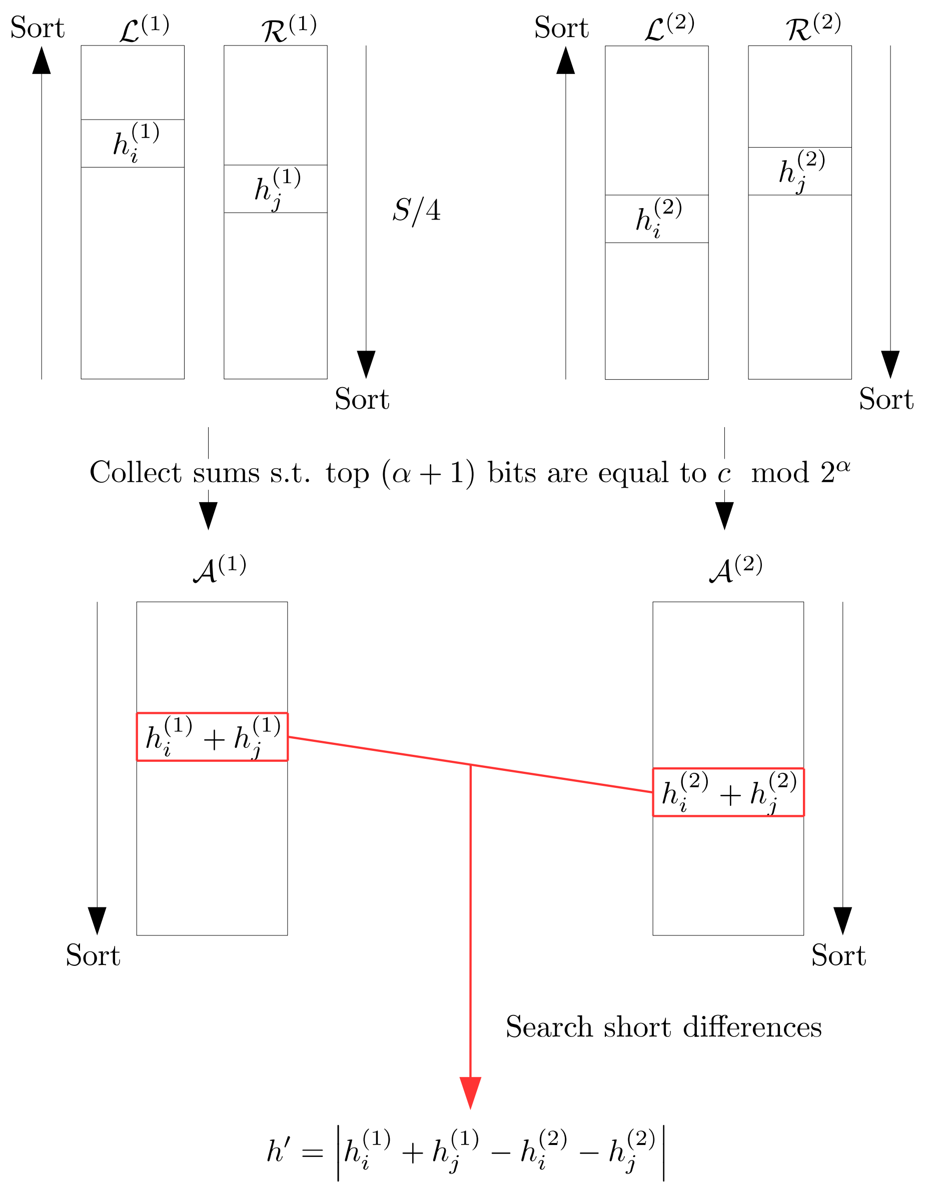


Figure 7: Overview of the Schroeppel–Shamir-based range reduction algorithm transformed from Howgrave-Graham–Joux's variant

Algorithm 3 Parallelizable Schroeppel–Shamir-based range reduction

```
Input:
 1:   sigs $:= \{(h_i, s_i)\}_{i=0}^{S-1}$ - the set of preprocessed Schnorr-like signatures with biased nonces
      $\lambda$ - bit-length of $h$, e.g., $\lambda = 252$ for qDSA signatures
      $\iota$ - number of iterations
      $\beta$ - number of bits to be reduced per round
Output: sols $:= \{(h'_j, s'_j)\}_{j=0}^{L-1}$ - a set of reduced signatures such that $h'_j < 2^{\lambda-\iota\beta}$
 2: $C \leftarrow 2^{\alpha}$
 3: $\tau \leftarrow \lambda$
 4: $L \leftarrow$ cardinality of sigs
 5: for $\rho = 1$ to $\iota$ do
 6:     Split sigs into 4 lists: $\mathcal{L}^{(1)}, \mathcal{R}^{(1)}, \mathcal{L}^{(2)}, \mathcal{R}^{(2)}$ of size $L' = L/4$
 7:     Sort $\mathcal{L}^{(1)}$ and $\mathcal{L}^{(2)}$ in descending order by $h$ values
 8:     Sort $\mathcal{R}^{(1)}$ and $\mathcal{R}^{(2)}$ in ascending order by $h$ values
 9:     Create empty lists sols, $\mathcal{A}^{(1)}$, and $\mathcal{A}^{(2)}$
10:     for $c = 0$ to $C - 1$ do
11:         Call Algorithm 4 on $\mathcal{L}^{(1)}, \mathcal{R}^{(1)}$, and $c$, push the result into a list $\mathcal{A}^{(1)}$
12:         Call Algorithm 4 on $\mathcal{L}^{(1)}, \mathcal{R}^{(1)}$, and $c + C$, push the result into a list $\mathcal{A}^{(1)}$
13:         Call Algorithm 4 on $\mathcal{L}^{(2)}, \mathcal{R}^{(2)}$, and $c$, push the result into a list $\mathcal{A}^{(2)}$
14:         Call Algorithm 4 on $\mathcal{L}^{(2)}, \mathcal{R}^{(2)}$, and $c + C$, push the result into a list $\mathcal{A}^{(2)}$
15:                 ▷ $\mathcal{A}^{(r)}$ is a list of $(\eta^{(r)}, \zeta^{(r)}) = \mathcal{L}^{(r)}[i] + \mathcal{R}^{(r)}[j] = (h_i^{(r)} + h_j^{(r)}, s_i^{(r)} + s_j^{(r)})$
16:         Sort $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$ in ascending order by $\eta$ values
17:         $i \leftarrow 0$
18:         $j \leftarrow 0$
19:         while Neither $\mathcal{A}^{(1)}[i]$ nor $\mathcal{A}^{(2)}[j]$ is at the end do
20:             if $\eta^{(1)}[i] > \eta^{(2)}[j]$ then
21:                 $(h', s') \leftarrow \mathcal{A}^{(1)}[i] - \mathcal{A}^{(2)}[j] = (\eta^{(1)}[i] - \eta^{(2)}[j], \zeta^{(1)}[i] - \zeta^{(2)}[j])$
22:                 Increment $j$
23:             else
24:                 $(h', s') \leftarrow \mathcal{A}^{(2)}[j] - \mathcal{A}^{(1)}[i] = (\eta^{(2)}[j] - \eta^{(1)}[i], \zeta^{(2)}[j] - \zeta^{(1)}[i])$
25:                 Increment $i$
26:             end if
27:             if $h' < 2^{\tau-\beta}$ then
28:                 Push $(h', s')$ to sols
29:             end if
30:         end while
31:     end for
32:     $\tau \leftarrow \tau - \beta$
33:     sigs $\leftarrow$ sols
34:     $L \leftarrow$ cardinality of sigs
35: end for
36: return sols
```

Algorithm 4 Collecting linear combinations of two routine

```
Input:
 1:      $\mathcal{L}$ - list of signatures sorted in descending order by $h$ values
         $\mathcal{R}$ - list of signatures sorted in ascending order by $h$ values
         $c$ - target value in a $(\alpha + 1)$-bit binary format
         $\tau$ - current bit-length upper bound of $h$ values
Output: $\mathcal{A}$ - list of linear combinations of two $(\eta, \zeta) = \mathcal{L}[i] + \mathcal{R}[j] = (h_i + h_j, s_i + s_j)$
    such that the value corresponding to the top consecutive $(\alpha + 1)$ bits of $\eta$ is equal to $c$,
    i.e., $\eta_{[\tau+1:\tau-\alpha+1]} = c$
 2: $i \leftarrow 0$
 3: $j \leftarrow 0$
 4: Create an empty list $\mathcal{A}$
 5: while Neither $\mathcal{L}[i]$ nor $\mathcal{R}[j]$ is at the end do
 6:     $(\eta, \zeta) \leftarrow \mathcal{L}[i] + \mathcal{R}[j] = (h_i + h_j, s_i + s_j)$
 7:     if $\eta_{[\tau+1:\tau-\alpha+1]} > c$ then
 8:         Increment $i$
 9:     else if $\eta_{[\tau+1:\tau-\alpha+1]} < c$ then
10:         Increment $j$
11:     else
12:         Push $(\eta, \zeta)$ to $\mathcal{A}$
13:         Peek at the value $\eta^{\mathcal{L}}$, where $(\eta^{\mathcal{L}}, \zeta^{\mathcal{L}}) := \mathcal{L}[i+1] + \mathcal{R}[j]$
14:         Peek at the value $\eta^{\mathcal{R}}$, where $(\eta^{\mathcal{R}}, \zeta^{\mathcal{R}}) := \mathcal{L}[i] + \mathcal{R}[j+1]$
15:         if $\eta^{\mathcal{L}}_{[\tau+1:\tau-\alpha+1]} = c$ and $\eta^{\mathcal{R}}_{[\tau+1:\tau-\alpha+1]} \neq c$ then
16:             Increment $i$
17:         else if $\eta^{\mathcal{L}}_{[\tau+1:\tau-\alpha+1]} \neq c$ and $\eta^{\mathcal{R}}_{[\tau+1:\tau-\alpha+1]} = c$ then
18:             Increment $j$
19:         else
20:             $k \leftarrow 1$
21:             if $\eta_{\tau-\alpha} = 0$ then
22:                 while $\eta^{\mathcal{L}}_{[\tau+1:\tau-\alpha+1]} = c$ do
23:                     Push $(\eta^{\mathcal{L}}, \zeta^{\mathcal{L}})$ to $\mathcal{A}$
24:                     Increment $k$
25:                     $(\eta^{\mathcal{L}}, \zeta^{\mathcal{L}}) \leftarrow \mathcal{L}[i+k] + \mathcal{R}[j]$
26:                 end while
27:                 Increment $j$
28:             else
29:                 while $\eta^{\mathcal{R}}_{[\tau+1:\tau-\alpha+1]} = c$ do
30:                     Push $(\eta^{\mathcal{R}}, \zeta^{\mathcal{R}})$ to $\mathcal{A}$
31:                     Increment $k$
32:                     $(\eta^{\mathcal{R}}, \zeta^{\mathcal{R}}) \leftarrow \mathcal{L}[i] + \mathcal{R}[j+k]$
33:                 end while
34:                 Increment $i$
35:             end if
36:         end if
37:     end if
38: end while
39: return $\mathcal{A}$
```

each c, the cardinality of the list $\mathcal{A}$ is estimated as follows:

$$\begin{aligned}|\mathcal{A}| &= L'^2 \cdot \Pr\left[\eta_{[\tau:\tau-\alpha+1]} = c\right] \\ &= C\end{aligned}$$

Now let us find the expected number of $(\tau-\beta)$-bit-bounded linear combinations of four. We would like to compute the following probability:

$$\begin{aligned}&\Pr\left[|\eta^{(1)} - \eta^{(2)}| < 2^{\tau-\beta} \middle| \eta^{(1)}_{[\tau:\tau-\alpha+1]} = \eta^{(2)}_{[\tau:\tau-\alpha+1]} = c\right] \\ =&\Pr\left[\eta^{(1)}_{\tau+1} = \eta^{(2)}_{\tau+1} \middle| \eta^{(1)}_{[\tau:\tau-\alpha+1]} = \eta^{(2)}_{[\tau:\tau-\alpha+1]} = c\right] \\ &\cdot \Pr\left[|\eta^{(1)}_{[\tau-\alpha:1]} - \eta^{(2)}_{[\tau-\alpha:1]}| < 2^{\tau-\beta} \middle| \eta^{(1)}_{[\tau:\tau-\alpha+1]} = \eta^{(2)}_{[\tau:\tau-\alpha+1]} = c\right]\end{aligned} \tag{3}$$

For notational simplicity, we will omit the condition event $\eta^{(1)}_{[\tau:\tau-\alpha+1]} = \eta^{(2)}_{[\tau:\tau-\alpha+1]} = c$ in the rest of the proof.

First, we compute the probability that $\eta^{(1)}_\tau$ and $\eta^{(2)}_\tau$ coincide:

$$\begin{aligned}&\Pr\left[\eta^{(1)}_{\tau+1} = \eta^{(2)}_{\tau+1}\right] \\ =&\Pr\left[\eta^{(1)}_{\tau+1} = 0\right] \cdot \Pr\left[\eta^{(2)}_{\tau+1} = 0\right] \\ +&\Pr\left[\eta^{(1)}_{\tau+1} = 1\right] \cdot \Pr\left[\eta^{(2)}_{\tau+1} = 1\right] \\ =&\left(\frac{c+1}{C}\right)^2 + \left(1 - \frac{c+1}{C}\right)^2\end{aligned}$$

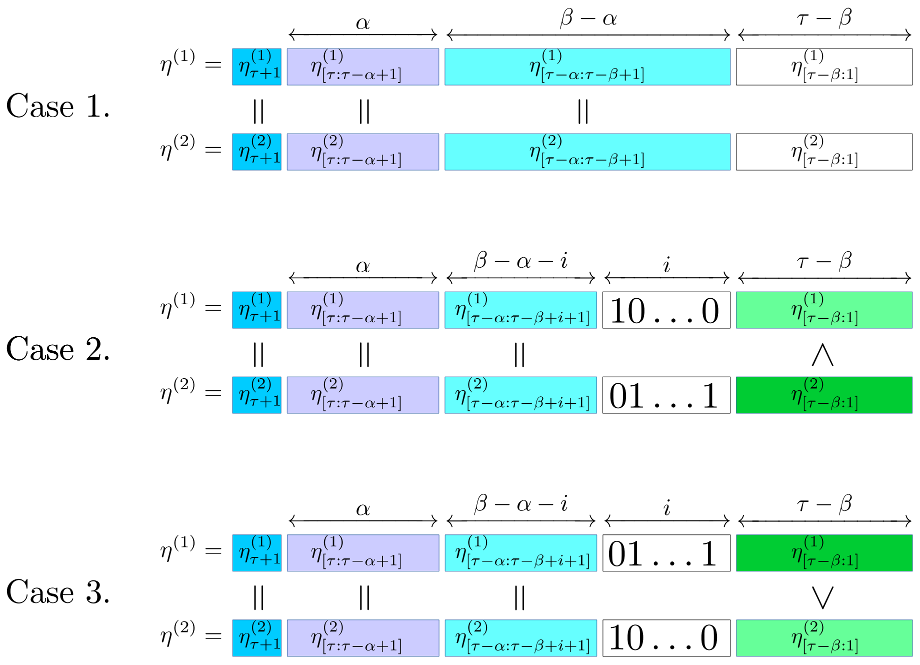
Second, we compute the following probability:

$$\Pr\left[|\eta^{(1)}_{[\tau-\alpha:1]} - \eta^{(2)}_{[\tau-\alpha:1]}| < 2^{\tau-\beta}\right]$$

We can consider three cases for the above, which are visualized in Figure 8.

Therefore, it can be computed as follows:

$$\begin{aligned}&\Pr\left[|\eta^{(1)}_{[\tau-\alpha:1]} - \eta^{(2)}_{[\tau-\alpha:1]}| < 2^{\tau-\beta}\right] \\ =&\Pr\left[\eta^{(1)}_{[\tau-\alpha:\tau-\beta+1]} = \eta^{(2)}_{[\tau-\alpha:\tau-\beta+1]}\right] \\ &+ \sum_{i=1}^{\beta-\alpha} \Pr\left[\eta^{(1)}_{[\tau-\alpha:\tau-\beta+i+1]} = \eta^{(2)}_{[\tau-\alpha:\tau-\beta+i+1]}\right] \cdot \\ &\Bigg(\Pr\left[\eta^{(1)}_{[\tau-\beta+i:\tau-\beta+1]} = \texttt{10...0} \wedge \eta^{(2)}_{[\tau-\beta+i:\tau-\beta+1]} = \texttt{01...1} \wedge \eta^{(1)}_{[\tau-\beta:1]} < \eta^{(2)}_{[\tau-\beta:1]}\right] \\ &+ \Pr\left[\eta^{(1)}_{[\tau-\beta+i:\tau-\beta+1]} = \texttt{01...1} \wedge \eta^{(2)}_{[\tau-\beta+i:\tau-\beta+1]} = \texttt{10...0} \wedge \eta^{(1)}_{[\tau-\beta:1]} > \eta^{(2)}_{[\tau-\beta:1]}\right]\Bigg) \\ =&\frac{1}{2^{\beta-\alpha}} + \sum_{i=1}^{\beta-\alpha} \frac{2}{2^{\beta-\alpha-i}}\left(\frac{1}{2^i} \cdot \frac{1}{2^i} \cdot \frac{1}{2}\right) \\ =&\frac{1}{2^{\beta-\alpha}} \sum_{i=0}^{\beta-\alpha} \frac{1}{2^i} \\ =&\frac{2}{2^{\beta-\alpha}} + \frac{2}{2^{2(\beta-\alpha)}}\end{aligned}$$



Case 1.
Case 2.
Case 3.
α
$\beta - \alpha$
$\tau - \beta$
$\beta - \alpha - i$
i
$\eta^{(1)} =$
$\eta^{(2)} =$
$\eta^{(1)}_{\tau+1}$
$\eta^{(1)}_{[\tau:\tau-\alpha+1]}$
$\eta^{(1)}_{[\tau-\alpha:\tau-\beta+1]}$
$\eta^{(1)}_{[\tau-\beta:1]}$
$\eta^{(1)}_{[\tau-\alpha:\tau-\beta+i+1]}$
10...0
01...1

Figure 8: Three cases where a small linear combination of four such that $h' = |\eta^{(1)} - \eta^{(2)}| < 2^{\tau-\beta}$ is found in Algorithm 3

Putting together, we can find the probability (3). Since there are $L'^4/C^2 = C^2$ possible linear combinations between $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$, for each $c \in [0, C)$, we obtain L_c linear combinations of four that are $(\tau - \beta)$-bit-bounded, where L_c is

$$L_c = C^2 \cdot \Pr\left[|\eta^{(1)} - \eta^{(2)}| < 2^{\tau-\beta}\right]$$

Finally, L, the expected cardinality of `sols`, is found by computing the sum of L_c:

$$\begin{aligned}
L =& \sum_{c=0}^{C-1} L_c \\
=& C^2 \left(\frac{2}{2^{\beta-\alpha}} + \frac{2}{2^{2(\beta-\alpha)}}\right) \sum_{c=0}^{C-1} \left\{ \left(\frac{c+1}{C}\right)^2 + \left(1 - \frac{c+1}{C}\right)^2 \right\} \\
=& C^2 \left(\frac{2}{2^{\beta-\alpha}} + \frac{2}{2^{2(\beta-\alpha)}}\right) \left(\frac{2}{3}C + \frac{1}{3C}\right) \\
=& \frac{4}{3}\left(2^{4\alpha-\beta} + 2^{5\alpha-2\beta}\right) + \frac{2}{3}\left(2^{2\alpha-\beta} + 2^{3\alpha-2\beta}\right)
\end{aligned}$$

Now we can directly derive the following claim.

Corollary 2. *After the first round of Algorithm 3, $L \approx S$ if $\beta = \beta_0 := 3\alpha - \log 3$.*

Proof.

$$\begin{aligned}
L &= \frac{4}{3}\left(2^{4\alpha-\beta_0} + 2^{5\alpha-2\beta_0}\right) + \frac{2}{3}\left(2^{2\alpha-\beta_0} + 2^{3\alpha-2\beta_0}\right) \\
&= \frac{4}{3} 2^{\alpha+\log 3} + o(1) \\
&\approx 4 \cdot 2^{\alpha} = S
\end{aligned}$$

After the first round, the above result does not hold strictly because h values are not uniformly distributed anymore. However, we empirically confirmed that approximately S reduced signatures can be constantly obtained in practice when β is sufficiently close to β_0. We first generated 2^{15} Schnorr signature pairs (h, s) over a group of 252-bit order, and then made Algorithm 3 reduce them for 5 times, i.e., the parameters were as follows: $S = 2^{15}$, $\lambda = 252$, and $\iota = 5$. Since $1.58 < \log 3 < 1.59$, we conducted the reduction experiments with $\beta = 3\alpha - 1.58$ and $\beta = 3\alpha - 1.59$ respectively, and measured the amount of reduced signatures after each iteration. Table 1 gives the experimental results. As a consequence, we actually managed to get more than S signatures after every round when $\beta = 3\alpha - 1.59$, which is slightly below β_0; on the other hand, the number of reduced signatures L decreased per iteration when $\beta = 3\alpha - 1.58 > \beta_0$. These results show that choosing β such that $\beta \leq \beta_0$ is indeed sufficient to maintain $L \approx S$ even after the first round (If the choice of β ends up with more than S reduced signatures, then we can simply interrupt the for loop as soon as the cardinality of `sols` reaches S, which of course makes the range reduction end faster). In what follows, we will assume that β is equal to or slightly smaller than β_0 to make the space usage stable.

Lemma 2. *The space complexity of Algorithm 3 is $O(S)$ if $\beta = \beta_0$.*

Proof. The space usage of Algorithm 3 is bound by the size of `sigs`, $\mathcal{A}^{(1)}, \mathcal{A}^{(2)}$ and `sols`, all of which have cardinality of at most $O(S)$ if $\beta = \beta_0$.

Lemma 3. *The time complexity of Algorithm 3 is $\widetilde{O}(S^2)$ if $\beta = \beta_0$.*

Table 1: Experimental results on the number of reduced signatures L after ρ rounds of the range reduction by Algorithm 3, when $\alpha = 15$ and $S = 2^{\alpha+2} = 131072$

	1	2	3	4	5
$\beta = 3\alpha - 1.59$	131343	132807	138622	160763	180003
$\beta = 3\alpha - 1.58$	130447	128226	120601	93524	34272

Table 2: Complexities and the number of reduced bits

Algorithm	Time	Space	Bits reduced
Ours (1 round)	$\widetilde{O}(S^2)$	$O(S)$	$3\alpha - \log 3$
Sort-and-difference (2 rounds)	$\widetilde{O}(S)$	$O(S)$	$2(\alpha + 2 - \gamma)$

Proof. We assume $L \approx S$ from Corollary 2. At the line 7 and 8, it takes time $O(S \log S)$ to sort the lists with a standard sorting algorithm such as quick sort. Collecting the linear combinations of two by Algorithm 4 takes $O(S)$ from the line 11 to 14. Since $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$ have the cardinality of $L'^2/C = S/4$, sorting at the line 16 takes $O(S \log S)$ and going through them in the while loop requires $O(S)$ steps for each c. We finally obtain $\widetilde{O}(S^2)$ by taking the summation from $c = 0$ to $c = C - 1$.

Table 2 gives the performance comparison between Algorithm 3 (with $\beta = \beta_0$) and the sort-and-difference assuming that both algorithms take the same input size S. Note that we evaluated 2 rounds of sort-and-difference for a fair comparison, since each iteration of it only constructs linear combinations of two, while our Schroeppel–Shamir-based algorithm constructs the linear combinations of four per round. Our approach can reduce more bits than the sort-and-difference per each equivalent round using the same amount of inputs; in other words, in order to reduce the same amount of bits, it takes less space complexity, and therefore requires fewer input signatures.

5.3 Parallelization

On the negative side, our algorithm takes more time complexity than the sort-and-difference. However, the large-scale parallelization of Algorithm 3 can compensate for it in practice. A careful reader may note that the procedure inside the for loop beginning at the line 10, which we call a *job* denoted by J_c with $c \in [0, C-1]$, is completely self-contained; in fact, a distributed-memory parallel computing allows us to implement the algorithm within a simple *master-worker* paradigm. That is, the master node simply broadcasts the sorted input data $(\mathcal{L}^{(1)}, \mathcal{R}^{(1)}, \mathcal{L}^{(2)}, \mathcal{R}^{(2)})$ and distributes the jobs $\{J_0, \ldots, J_{C-1}\}$ evenly to worker nodes, so the workers can focus on their own small set of jobs independently, i.e., without communicating with other workers. In Section 6, we will revisit the parallelization setting and describe concrete implementation techniques.

5.4 Lower Bounds for the Amount of Signatures

As we observed in Section 4, a range reduction algorithm needs to output *small* and *sparse* linear combinations as specified in **[C1]** and **[C2]** of Bleichenbacher's attack framework (Algorithm 2). Given group order bit-length λ, biased MSB in a nonce b and the rounds of reduction ι as parameters, we can derive the lower bounds for the amount of input signatures for our range reduction algorithm to satisfy the two conditions.

Theorem 2. *Algorithm 3 satisfies* **[C1]** *and* **[C2]** *after ι rounds if $\beta = \beta_0$ and $S = 2^{\alpha+2} > 2^{\alpha_{\mathrm{SS}}+2}$, where*

$$\alpha_{\mathrm{SS}} = \max\left\{\frac{\lambda - 2 + \iota\log 3}{1+3\iota}, 2\cdot 4^{\iota}\left(\log\pi - b - \log\sin(\pi/2^b)\right)\right\}$$

Proof. From Corollary 2, we assume $L = S$. Since after ι rounds of range reduction, we get linear combinations such that $h' < 2^{\lambda-\iota\beta_0}$. Hence, to satisfy **[C1]**,

$$\begin{aligned}
&2^{\lambda-\iota\beta_0} \le S\\
\Leftrightarrow&\lambda - \iota\beta_0 \le \log S = \alpha + 2\\
\Leftrightarrow&\alpha \ge \frac{\lambda - 2 + \iota\log 3}{1+3\iota}
\end{aligned}$$

Algorithm 3 constructs linear combinations of four per each round, i.e., it creates $h' = |\eta^{(1)} - \eta^{(2)}| = |h^{(1)}_{i_1} + h^{(1)}_{j_1} - h^{(2)}_{i_2} - h^{(2)}_{j_2}|$ and its corresponding nonce becomes $k' = |k^{(1)}_{i_1} + k^{(1)}_{j_1} - k^{(2)}_{i_2} - k^{(2)}_{j_2}|$.

Recalling the discussion in Section 4.2, we can approximate the resulting bias as follows: Let $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}, \boldsymbol{W}$ be uniformly distributed and independent random variables on $[0, n-1]$. Then applying (b) and (c) of Lemma 1, $B_n(\boldsymbol{X}) = B_n(\boldsymbol{Y}) = B_n(\boldsymbol{Z}) = B_n(\boldsymbol{W})$ and

$$\begin{aligned}
B_n(\boldsymbol{X}+\boldsymbol{Y}-\boldsymbol{Z}-\boldsymbol{W}) &= B_n(\boldsymbol{X}+\boldsymbol{Y})\cdot\overline{B_n(\boldsymbol{Z}+\boldsymbol{W})}\\
&= B_n(\boldsymbol{X})\cdot B_n(\boldsymbol{Y})\cdot\overline{B_n(\boldsymbol{Z})\cdot B_n(\boldsymbol{W})}\\
&= B_n(\boldsymbol{X})\cdot B_n(\boldsymbol{Y})\cdot\overline{B_n(\boldsymbol{Z})}\cdot\overline{B_n(\boldsymbol{W})}\\
&= |B_n(\boldsymbol{X})|^4
\end{aligned}$$

This means that each iteration approximately reduces the bias by raising it to the fourth power [4]. Therefore, the condition **[C2]** can be rewritten as follows:

$$|B_{n,b}(\boldsymbol{K})|^{4^{\iota}} > \underbrace{1/\sqrt{S}}_{\text{size of noise}}$$

Applying Corollary 1, we obtain

$$\begin{aligned}
&\left(\frac{2^b}{\pi}\sin(\pi/2^b)\right)^{4^{\iota}} > 1/\sqrt{S} = 1/\sqrt{2^{\alpha+2}}\\
\Leftrightarrow&4^{\iota}\left(b - \log\pi + \log\sin(\pi/2^b)\right) + \frac{1}{2}(\alpha+2) > 1\\
\Leftrightarrow&\alpha > 2\cdot 4^{\iota}\left(\log\pi - b - \log\sin(\pi/2^b)\right)
\end{aligned}$$

Putting together, we obtain α_{SS}.

5.5 Data-(Time, Space) Trade-off

In practice, adversaries who can perform the fault attack are allowed to generate as many signatures as they want and filter out ones with relatively large h. That is, let f be the number of bits to be filtered, then one can heuristically get S signatures such that

[4] Again, the argument here is not rigorous since there is possibility after the first round that the same original h values are used more than once in a resulting linear combination. However, the approximation holds in practice just as [AFG+14] confirmed so for the sort-and-difference.

$h < 2^{\lambda - f}$ by generating $2^f \cdot S$ faulty signatures, assuming that h is uniformly distributed in $[0, 2^\lambda - 1]$. With this setting, the condition [C1] is relaxed as follows:

$$\begin{aligned} &2^{\lambda - f - \iota\beta_0} \leq S \\ \Leftrightarrow &\lambda - f - \iota\beta_0 \leq \log S = \alpha + 2 \\ \Leftrightarrow &\alpha \geq \frac{\lambda - f - 2 + \iota \log 3}{1 + 3\iota} \end{aligned}$$

This clearly improves the lower bound obtained in Theorem 2 in exchange for spending more time on the initial signature generation. Let α'_{SS} be the new lower bound, then

$$\alpha'_{\mathrm{SS}} = \max\left\{ \frac{\lambda - f - 2 + \iota \log 3}{1 + 3\iota}, 2 \cdot 4^\iota \left(\log \pi - b - \log \sin(\pi/2^b)\right) \right\}$$

Now we only need to pass Bleichenbacher's attack at least $2^{\alpha'_{\mathrm{SS}}+2}$ signatures. Let $\mathcal{T}_{\mathsf{Gen}}$ be the time spent on signature generation, and $\mathcal{T}_{\mathsf{Atk}}$ and $\mathcal{S}_{\mathsf{Atk}}$ be the time and space required for Bleichenbacher's attack with our range reduction (i.e., Algorithm 2 & 3), respectively. Then, we obtain the following estimates for each:

$$\begin{aligned} \mathcal{T}_{\mathsf{Gen}} &\propto 2^{\alpha'_{\mathrm{SS}}+2+f} \\ \mathcal{T}_{\mathsf{Atk}} &\propto 2^{2(\alpha'_{\mathrm{SS}}+2)} \\ \mathcal{S}_{\mathsf{Atk}} &\propto 2^{\alpha'_{\mathrm{SS}}+2} \end{aligned}$$

Thus, the parameter f gives us the flexibility and it can be determined depending on the precise context; for example, if we are allowed to generate significantly many signatures, but can only utilize relatively limited computational resources, then f should be increased so as to obtain the appropriate lower bound α'_{SS}, and vice versa. We make use of this technique to attack 2-bit bias in Section 6.

5.6 Performance Comparison

We apply the settings of our attack — the qDSA on Curve25519 with its 2- or 3-LSB of the nonces known via fault attacks — to the bound obtained in Theorem 2 in order to give concrete performance estimates of our range reduction algorithm. We also found the optimal number of iterations ι for both cases such that α_{SS} is minimized. Table 3 summarizes the result. It includes the comparison with the sort-and-difference used by [AFG+14] and with a lattice attack in combination with the SVP algorithm by [BDGL16]. Note that the estimates for the sort-and-difference are too optimistic because they are based on the assumption that the ratio $(1 - e^{-2^\gamma})^\iota$ holds even after the first iteration; indeed, unlike our algorithm, it is not true in practice as we reviewed in Section 4. We actually encountered such a situation and acquired less resulting signatures than theoretically estimated (see Section 6).

6 Implementation Results

We implemented the Bleichenbacher's attack incorporating the reduction technique described in Algorithm 3. In this section, we summarize the implementation details and our experimental results. The source code of the programs used in this section is submitted as supplementary material.

Table 3: Estimates for the minimum required number of signatures and the optimal performance of reduction algorithms and a lattice attack when $\lambda = 252$

	Algorithm	ι	S	Time	Space
$b = 2$	Ours	3	$2^{27.5}$	$2^{55.0}$	$2^{27.5}$
	Sort-and-difference ($\gamma = 1$)	6	$2^{37.0}$	$2^{37.0}$	$2^{37.0}$
	Lattice attack	–	$2^{7.2}$	$2^{58.8}$	$2^{58.8}$
$b = 3$	Ours	4	$2^{21.7}$	$2^{43.4}$	$2^{21.7}$
	Sort-and-difference ($\gamma = 1$)	8	$2^{29.1}$	$2^{29.1}$	$2^{29.1}$
	Lattice attack	–	$2^{6.5}$	$2^{43.3}$	$2^{43.3}$

Tools. We artificially generated faulty qDSA signatures by modifying the C reference implementation [Ren17a]. The attack program was written in C++ and the multiprecision integer arithmetic was mostly handled by GMP library [Gt16], except that the reduction phase only made use of the built-in C integer type `uint64_t` for more optimization; in fact, we do not need to handle the full-fledged big integers there since our reduction algorithm only requires the evaluation of the top β-bit and the following few bits, as Figure 8 depicts. The bias was computed with FFTW [FJ05]. The large-scale parallelization was achieved with the combination of Open MPI [GFB+04] and OpenMP [Ope08].

Hybrid shared/distributed-memory parallelization. We describe how the large-scale parallelization of Algorithm 3 was achieved in practice. We implemented the attack using hybrid shared-memory and distributed-memory parallel programming technique. The former was handled by OpenMP and the latter was by MPI.

We utilized the following two parallel computing facilities during the experiments:

1. a dual Xeon E5-2697 v3-based workstation (2 CPUs $\times$ 14 cores/CPU $\times$ 2 threads/-core), and

2. virtual machine instances on a distributed cluster (16 virtual machine nodes $\times$ 16 vCPU/node).

In particular, the much larger second facility is a distributed-memory system that consists of a set of independent nodes, each of which has its own shared-memory multiprocessing environment. (And although the first system is a single workstation with a single memory space, MPI also made it appear as though it consisted of two separate nodes running distinct multithreaded processes).

As a parallel programming paradigm, we employed a simple *master-worker* scheme (see, e.g., [HW11, Chapter 5] for details). Let t be the number of available shared-memory threads within a node and N be the number of distributed-memory nodes, where N is a power of 2 for simplicity. Moreover, we assume that each node is assigned a unique identifier $I \in [0, N-1]$. Then our parallelization strategy is summarized as follows:

1. Make the *master* process load and sort the input data

2. Map one MPI *worker* process per node

3. Broadcast the data, partition the set of jobs $\{J_0, \dots, J_{C-1}\}$ into N subsets $\mathcal{J}_0, \dots, \mathcal{J}_{N-1}$ of the same cardinality, assign node I a subset $\mathcal{J}_I$

4. Make the workers spawn a team of t OpenMP threads inside and process the assigned jobs

5. Gather the results (i.e., subsets of `sols`) into the master

To achieve these, calling a few basic MPI collective communication routines — `MPI_Bcast`, `MPI_Gather`, and `MPI_GatherV` — is sufficient. Each routine was called only once per round before/after the for loop and it only took a few minutes to broadcast and gather the data in both experiments below. Considering the time spent on the whole range reduction operations, our implementation introduces negligibly low communication overhead due to the parallelization.

Scalability. Although our range reduction algorithm is highly space-efficient, multi-threading in a shared-memory environment requires extra space for storing the lists $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$, whose expected cardinalities are $C = S/4$, for each thread (see the proof of Theorem 1). On the other hand, the amount of distributed-memory nodes N divides the cardinality of `sols` stored in each node. Therefore, the space needed for each node can be roughly estimated as follows:

$$\underbrace{S}_{\mathcal{L}^{(1)},\mathcal{R}^{(1)},\mathcal{L}^{(2)},\mathcal{R}^{(2)}} + \underbrace{2tC}_{\mathcal{A}^{(1)},\mathcal{A}^{(2)}} + \underbrace{S/N}_{\text{(partial) }\texttt{sols}}$$

Recalling the fact that our implementation broadcasts and gathers the data between nodes *only once*, it is advisable to scale distributed-memory nodes instead of shared-memory threads to save the memory space. In the era of cloud computing, it is safe to say that preparing many distributed nodes with moderate memory capacity is not very difficult for well-funded adversaries in practice. Hence, our range reduction algorithm is highly scalable. In the following subsection, we will show our measured performance results on the virtual distributed-memory instances on the cluster machine (i.e., N=16 and t=16).

6.1 Attack against 2-bit Bias

We first present our main result: the key recovery attack against qDSA instantiated with Curve25519 using 2-bit biased nonces. We artificially generated faulty qDSA signatures based on the fault attack described in Section 3.2; in addition, we preprocessed them to make 2-MSB of nonces biased as described in Section 3.3. Due to the computational resources available to us, we had to filter the signature pairs by h values to trade the time and space complexity for the data complexity, following the discussion in Section 5.5. More concretely, we set $f = 19$ and initially generated nearly 2^{45} preprocessed signatures and only kept ones such that $h < 2^{252-19}$, so that we obtained $S = 2^{26}$ signatures to be processed by Bleichenbacher's attack. Accordingly, we only had to reduce $252 - 19 - 26 = 207$-bit in total during the range reduction phase, which allowed us to set the parameter $\beta = 69$ slightly below $\beta_0 = 72 - \log 3$.

The recovery of the first MSB was conducted with the virtual machine instances, i.e., the range reduction jobs were distributed to 16 distributed-memory MPI processes, all of which spawned 16 shared-memory OpenMP threads. The measured performance results are summarized in Table 4. We observed that the detected bias peak after 3 rounds of reduction matches the theoretical estimate, i.e., using Corollary 1, $|B_{n,2}(\boldsymbol{K})|^{4^3} \approx 0.0012$. The detected sampled biases are plotted in Figure 9. (It only displays the selected noise points for simplicity; we actually computed the sampled biases at L-evenly-spaced points in n, where $L \approx 2^{26}$, and detected the only one peak point that showed the significant bias value.) The FFT table preparation and sampled bias computation finished within a few minutes.

Though the total wall clock time was over two weeks, we expect much better performance on a dedicated cluster. Due to the uneven resource allocation of virtual instances, which are used by many people and therefore out of our control, some nodes were significantly slower than others, and the fastest node completed their jobs within only 7 days, which is equivalent to 4.8 CPU-years in total. As a matter of fact, we did not observe such a difference when we parallelized the range reduction on the Xeon workstation. Thus, we stress that this synchronization overhead is not because of our range reduction algorithm, but rather a specific problem in virtual machines.

After the 26-MSB of the secret key was successfully recovered, we iteratively recovered the following bits as in Section 4.4, using the 2 nodes (i.e., 56 threads in total) of the Xeon workstation for the range reduction. Consequently, the whole process below took less than 6 hours in total. We took a small security margin and only assumed that the 24-MSB was recovered in the previous phase, following the advice by [AFG$^+$14] and [DMHMP14]. We used Algorithm 3 until we recovered the 189-MSB and lastly used the sort-and-difference to recover the 216-MSB; at this stage, we do not need to reduce many bits anymore, and therefore the sort-and-difference is more convenient since it only constructs linear combinations of two and does not diminish the sampled bias peak very much, which allows us to detect the peak area more precisely. Finally, we directly computed the bias without range reduction and recovered 241-MSB, with which a simple exhaustive search could be easily done to obtain the rest of the secret.

Performance estimate of better-equipped adversaries. Since we filtered input signatures , what we have computed corresponds to the time $\mathcal{T}_{\mathsf{Atk}}$ of 2^{52} and the space $\mathcal{S}_{\mathsf{Atk}}$ of 2^{26}, which are approximated based on the data-(time, space) trade-off discussed in Section 5.5. Thus, we can infer that a better-equipped adversary, say one with access to 32 cores $\times$ 32 nodes with 96GB RAM for each, could perform a key recovery within about 3 months even without filtering at all, from the estimate in Table 3. This should be a more favorable attack setting in a situation where the adversary is only allowed to generate faulty signatures in constrained embedded systems, e.g., IoT devices. Hence, we conclude that a key recovery attack against 252-bit signatures with 2-bit nonce bias could even be achievable in a real-world scenario using the combination of Bleichenbacher's attack framework and our range reduction algorithm.

6.2 Attack against 3-bit Bias

Next, we describe the result of the attack against qDSA signatures with 3-bit biased nonces. We artificially generated 2^{23} faulty signatures (without filtering) based on the attack in Section 3.1 and preprocessed them to make the 3-MSB of nonces biased as described in Section 3.3. The program was executed in the Xeon workstation and we parallelized the range reduction with 28 shared-memory OpenMP threads $\times$ 2 MPI nodes.

The measured performance results are given in Table 5. We also performed the attack using the sort-and-difference, which is abbreviated as S&D. The attack was completed much faster than the case of 2-bit bias since now we are allowed to iterate the range reduction 4 times, and therefore the amount of bits reduced per round is much less. Moreover, the CPU-time was almost 10 days and the memory consumption was considerably lower then that of the sort-and-difference. This result implies that the attack against 3-bit bias would even be feasible using a small laptop for daily use. We omit the recovery of the following bits since the procedure is the same as the previous experiment on 2-bit bias.

It also turned out that the sort-and-difference (with $\gamma = 1$) is even exploitable against 3-bit bias and the CPU-time was a lot shorter than our algorithm, which is as expected. In a situation where an adversary is allowed to generate more than 1 billion 3-bit biased signatures, the use of sort-and-difference should be a better option. However, it should be pointed out that the resulting number of signatures after 8 rounds was only $2^{25.8}$, which is significantly less than the estimated amount, i.e., $(1 - e^{-2^{\gamma}})^8 \cdot 2^{30} \approx 2^{28.3}$. This instability could be an obstacle when attacking the signatures over a bigger group, since it demands larger γ or more input signatures than the theoretical bound, either of which would lead to more memory usage than expected.

Table 4: Implementation results of the attack against qDSA signatures with nonces of 2-bit bias

Wall clock time	CPU-time	Memory	ι	S	Peak	#Recovered MSB
400.7 hours	11.7 years	15GB	3	2^{26}	0.0012	26-bit

Table 5: Implementation results of the attack against qDSA signatures with nonces of 3-bit bias

	Wall clock time	CPU-time	Memory	ι	S	Peak	#Recovered MSB
Ours	4.25 hours	238 hours	2.8GB	4	2^{23}	0.0016	23-bit
S&D	0.75 hours	0.75 hours	128GB	8	2^{30}	0.0014	21-bit

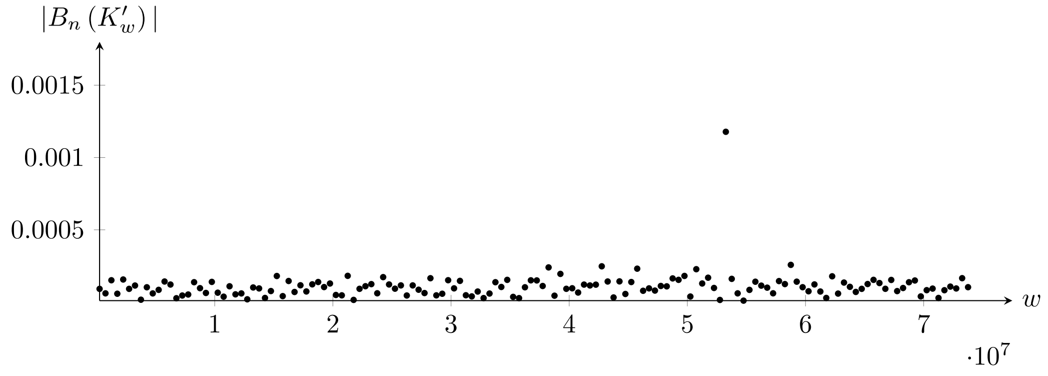


Figure 9: Detected sampled biases after reducing the signatures with 2-bit biased nonces 3 times

7 Conclusion

In this paper, we have proposed fault attack techniques against the qDSA signature scheme to induce a few bits bias in the nonces. Furthermore, we designed a highly-parallelizable and space-efficient range reduction algorithm for the Bleichenbacher's nonce attack, based on Schroeppel–Shamir algorithm. We presented the first complete experimental results on the full key recovery of 252-bit curve with 2-bit and 3-bit biased nonces, and thus have set the new records of Bleichenbacher's attack.

References

[AFG$^+$14] Diego F. Aranha, Pierre-Alain Fouque, Benoit Gérard, Jean-Gabriel Kammerer, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. GLV/GLS decomposition, power analysis, and attacks on ECDSA signatures with single-bit nonce bias. In T. Iwata and P. Sarkar, editors, *ASIACRYPT 2014*, volume 8873 of *LNCS*, pages 262–281. Springer, 2014.

[BCN$^+$06] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, *SODA*, pages 10–24. SIAM, 2016.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In *EUROCRYPT '97*, volume 1233 of *LNCS*, pages 37–51. Springer, 1997.

[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, 2012.

[Ber06] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006.

[BFMT16] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. Side-channel analysis of weierstrass and koblitz curve ECDSA on android smartphones. volume 9610 of *LNCS*, pages 236–252. Springer, 2016.

[BL] Daniel J. Bernstein and Tanja Lange. SafeCurves: choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to.

[BL17] Daniel J. Bernstein and Tanja Lange. Montgomery curves and the montgomery ladder. Cryptology ePrint Archive, Report 2017/293, 2017. http://eprint.iacr.org/2017/293.

[Ble00] Daniel Bleichenbacher. On the generation of one-time keys in DL signature schemes. Presentation at IEEE P1363 working group meeting, 2000. Available from http://grouper.ieee.org/groups/1363/Research/contributions/Ble2000.tif.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. Differential fault attacks on elliptic curve cryptosystems. In *CRYPTO 2000*, volume 1880 of *LNCS*, pages 131–146. Springer, 2000.

[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. Springer, 1996.

[BvdPSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "Ooh aah... just a little bit" : A small amount of side channel can go a long way. In *CHES 2014*, volume 8731 of *LNCS*, pages 75–92. Springer, 2014.

[CS17] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic: The case of large characteristic fields. Cryptology ePrint Archive, Report 2017/212, 2017. http://eprint.iacr.org/2017/212.

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

[DMHMP14] Elke De Mulder, Michael Hutter, Mark E Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version. *Journal of Cryptographic Engineering*, 4(1):33–45, 2014.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[FFAL17] Armando Faz-Hernández, Hayato Fujii, Diego F. Aranha, and Julio López. A secure and efficient implementation of the quotient digital signature algorithm (qdsa). In Sk Subidh Ali, Jean-Luc Danger, and Thomas Eisenbarth, editors, *SPACE 2017*, volume 10662 of *LNCS*, pages 170–189. Springer, 2017.

[FJ05] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[FLRV08] Pierre-Alain Fouque, Reynald Lercier, Denis Réal, and Frédéric Valette. Fault attack on elliptic curve Montgomery ladder implementation. In Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *FDTC 2008*, pages 92–98. IEEE, 2008.

[Gal13] Patrick Gallagher. *Digital signature standard (DSS)*. NIST, 2013. FIPS PUB 186–4.

[GFB$^+$04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[Gt16] Torbjörn Granlund and the GMP development team. *GMP: The GNU Multiple Precision Arithmetic Library Version 6.1.2*, 2016. http://gmplib.org/.

[GVY17] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *CCS 2017*, pages 845–858. ACM, 2017.

[HGJ10] Nick Howgrave-Graham and Antoine Joux. New generic algorithms for hard knapsacks. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 235–256. Springer, 2010.

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

[HW11] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman and Hall / CRC computational science series. CRC Press, 2011.

[KS00] David R. Kohel and Igor E. Shparlinski. On exponential sums and group generators for elliptic curves over finite fields. In *ANTS-IV*, pages 395–404, 2000.

[LHT16] Adam Langley, Mike Hamburg, and Sean Turner. *Elliptic curves for security*. IRTF, 2016. RFC 7748.

[LN13] Mingjie Liu and Phong Q. Nguyen. Solving BDD by enumeration: An update. In *CT-RSA 2013*, volume 7779 of *LNCS*, pages 293–309. Springer, 2013.

[Mac] Kenneth MacKay. micro-ecc: a small and fast implementation of ECDSA and ECDH for 8-bit, 32-bit, and 64-bit processors. https://github.com/kmackay/micro-ecc.

[Mon87] Peter L Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[NNTW05] David Naccache, Phong Q. Nguyen, Michael Tunstall, and Claire Whelan. Experimenting with faults, lattices and the DSA. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 16–28. Springer, 2005.

[NS02] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *Journal of Cryptology*, 15(3), 2002.

[NS03] Phong Q Nguyen and Igor E Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, Codes and Cryptography*, 30(2):201–217, 2003.

[NT12] Phong Q. Nguyen and Mehdi Tibouchi. *Lattice-Based Fault Attacks on Signatures*, pages 201–220. Information Security and Cryptography. Springer, 2012.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *COSADE*, volume 8622 of *LNCS*, pages 243–260. Springer, 2014.

[Ope08] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[Ren17a] Joost Renes. qDSA reference implementation for C. https://www.cs.ru.nl/~jrenes/software/cref-g1.tar.gz, 2017.

[Ren17b] Joost Renes. qDSA reference implementation for the AVR ATmega. https://www.cs.ru.nl/~jrenes/software/avr-g1.tar.gz, 2017.

[RS17] Joost Renes and Benjamin Smith. qDSA: Small and secure digital signatures with curve-based Diffie-Hellman key pairs. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017*, volume 10625 of *LNCS*, pages 273–302. Springer, 2017.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[SS81] Richard Schroeppel and Adi Shamir. A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal on Computing*, 10(3):456–464, 1981.