Batch Verification of Modified ECDSA Signatures

İrem Keskinkurt Paksoy[1][0000−0003−2529−1150], Muhammad ElSheikh[1,2][0000−0003−3678−6140], Murat Cenk[3][0000−0003−4941−8734], and M. Anwar Hasan[1][0000−0003−4103−7945]

[1] University of Waterloo, Waterloo ON N2L 3G1, Canada
{ikeskink,mhgaelsh,ahasan}@uwaterloo.ca
[2] National Institute of Standards, Giza, 12211, Egypt
[3] Ripple Labs Inc, San Francisco, CA 94111, USA
mcenk@ripple.com

Abstract. ECDSA is a widely deployed digital signature scheme used in securing communications and distributed systems but its structure prevents efficient batch verification. This limitation is impactful in high-throughput applications such as blockchain transaction validation. In this paper, we consider two modified versions of standard ECDSA, namely ECDSA_rec and ECDSA_ast, which embed additional bits in the signature, that enable efficient batch verification. We contribute the first publicly available C implementations of ECDSA_ast on secp256k1 and of both ECDSA_ast and ECDSA_rec on secp256r1, built on top of Bitcoin and OpenSSL codebases, respectively. We evaluate both variants on the two curves and demonstrate batch verification using the two established randomization techniques, NMVR-rand and HSS-rand. The results of our software implementation indicate that batch verification of modified ECDSA signatures using HSS-rand generally outperforms individual verification of standard ECDSA. Furthermore, the HSS-rand technique provides more efficient batch verification than NMVR-rand in the majority of cases. For a batch of 32 ECDSA_ast (ECDSA_rec resp.) signatures on secp256k1, HSS-rand batch verification improves individual verification by approximately 30.9% (17.2% for ECDSA_rec) and outperforms NMVR-rand batch verification, which achieves improvements of 10.6% (9.8% for ECDSA_rec). The improvements achieved by HSS-rand on secp256r1 are even more pronounced, reaching 53.9% for ECDSA_ast and 40.5% for ECDSA_rec over individual verification of standard ECDSA signatures. In comparison, HSS-rand yields improvements of 27.6% for ECDSA_ast and 22.9% for ECDSA_rec over NMVR-rand, respectively.

Keywords: ECDSA· modified ECDSA · batch verification · signature optimization · half-size scalar multiplication · blockchain.

1 Introduction

Elliptic curve (EC) digital signatures are widely used to ensure authenticity and integrity in secure communications and distributed systems, including blockchains. In high-throughput environments, verifying large numbers of signatures incurs

significant computational cost. This overhead can be reduced by batch verification, a technique that aggregates multiple signatures for simultaneous verification. When supported by the signature scheme, batch verification can substantially improve system performance.

Elliptic Curve Digital Signature Algorithm (ECDSA) is the EC variant of the Digital Signature Standard and has been standardized by various bodies including the National Institute of Standards and Technology (NIST) [17]. The smaller key sizes of ECDSA, for the same level of security, make it more favorable for real-world applications. Unlike Schnorr-like signatures, such as the Edwards-curve Digital Signature Algorithm (EdDSA) [10], ECDSA signatures do not possess a structure that enables efficient batch verification. In [12, 13], modified formulas with additional field multiplications for doubling and addition were proposed to address this problem. Furthermore, [11] introduces the use of summation polynomials for the same purpose. While these works report improvements, none offer a technique efficient enough for practical cryptographic applications. The results in [11–13] are based on implementations in the GP/PARI language, which is well suited for symbolic computation. Since their software has not been made publicly available, reproducing the results is not possible; therefore, their results cannot be readily compared with more recent studies.

In [1], two variants of standard ECDSA signatures were introduced and we refer to them as ECDSA_rec and ECDSA_ast (originally referred to as ECDSA*). These variants' signatures incorporate additional bits into the standard signature thereby rendering batch verification more feasible. Although these variants are not standardized, they are still conformant with the existing ECDSA standards [1] and ECDSA_rec is included in the Bitcoin implementation for cases where public key recovery is required. While ECDSA_rec signatures add only a few extra bits, ECDSA_ast signatures are approximately 50% larger than those of standard ECDSA. Nevertheless, the ECDSA_ast scheme may be preferred in applications where efficient batch verification is a priority.

Unlike standard ECDSA, Schnorr-like signatures inherently work well with batch verification as they include both coordinates of the point rather than just the x-coordinate. The highly optimized ed25519-donna implementation of EdDSA [15] achieves efficient batch verification using a randomization technique introduced in [16] by Naccache, M'RaÏhi, Vaudenay and Raphaeli, and referred to as NMVR-rand in this paper. A recent study [7] proposed a division-free half-way extended Euclidean algorithm (hEEA_approx_q) for half-size scalar (HSS) generation and a novel randomization method based on it. We refer to the HSS based randomization method as HSS-rand. This method was applied to EdDSA batch verification and compared against NMVR-rand with ed25519-donna, demonstrating significant improvements over the already well-studied and highly optimized implementation [15].

Contributions: In this work, we apply, for the first time, the HSS-rand technique to the batch verification of modified ECDSA signatures. Moreover, we describe potential weaknesses in both batch verification techniques that arise from randomizer predictability. We implemented ECDSA_ast on secp256k1, and both

ECDSA_rec and ECDSA_ast on secp256r1. Our contributions also include the implementation of NMVR-rand and HSS-rand batch verification techniques for ECDSA_rec and ECDSA_ast, along with comparative benchmark results. Our implementations are extensions to the Bitcoin [4] and OpenSSL [18] ECDSA codebases, which utilize the secp256k1 and secp256r1 curves, respectively. To support reproducibility and facilitate further research, we make our software publicly available at the following repositories: https://github.com/iremkp/ossl-ecdsa.git and https://github.com/iremkp/bitcoin_ecdsa.git. To the best of our knowledge, our software represents the first publicly available implementation in the C programming language for batch verification of ECDSA variants.

The remainder of the paper is organized as follows. In Section 2, we provide some preliminaries necessary for a better understanding of the subsequent sections. In Section 3, we describe how batch verification is applied to the modified ECDSA schemes. In Section 4, we analyze the potential vulnaribilities that arise if the random values are known to an attacker. Implementation details and benchmark results are presented in Section 5. The performance results of our Bos–Coster implementation for HSS-rand batch verification of modified ECDSA signatures are presented in Section 6. Finally, Section 7 concludes the paper with a summary and final remarks.

2 Preliminaries

We use the notations $(R)_x$ and $(R)_y$ to denote the x- and y-coordinates of a point R on an EC. For an EC defined over a prime field $\mathbb{F}_p$, the prime order of the EC group will be denoted by n. We refer to integers with bit sizes up to that of n as *full-size* scalars, and those with bit sizes up to half that of n as *half-size* scalars. We use the notation $\mathcal{O}$ to denote the point at infinity on an elliptic curve.

Curves: We use secp256k1 and secp256r1 curves for our experiments. While both curves are in short Weierstrass form, secp256k1 is a Koblitz-like curve over a prime field that supports an efficient endomorphism [9] for fast scalar multiplication. secp256k1 has been standardized by SECG (Standards for Efficient Cryptography Group) [20], and is used in various blockchain implementations such as Bitcoin and Ethereum. On the other hand, secp256r1, also known as P-256, has been standardized by both SECG and NIST, among other bodies and is widely used across various applications and protocols such as Secure Web Communications (TLS/SSL) and VPN Protocols (IPsec). Unlike secp256k1, the secp256r1 curve, however, does not admit an endomorphism that can be exploited to accelerate scalar multiplication.

ECDSA: In a standard ECDSA signature $\sigma = (r, s)$, the value r is obtained from the x-coordinate of a randomly generated elliptic curve point $R = kB$, where $k \in (0, n)$ is a random nonce and B is the group generator. Since the x-coordinate

$(R)_x$ of R is an element of the base field $\mathbb{F}_p$, while n is the order of the subgroup generated by B, $(R)_x$ is reduced modulo n to ensure that r lies in the correct range. Thus, we have $r = (R)_x \bmod n$. Consequently, while each point R yields a unique value of r, the reverse does not hold: a given r may correspond to multiple valid points R on the curve. This ambiguity arises both from the fact that distinct points on the elliptic curve can share the same x-coordinate due to symmetry about the x-axis, and from the reduction of $(R)_x$ modulo n.

For an ECDSA signature $\sigma = (r, s)$ on a message m with a verification key A, one computes

$$R = uB + vA, \tag{1}$$

where $u = \left(\frac{H(m)}{s} \bmod n\right)$ and $v = \left(\frac{r}{s} \bmod n\right)$ for a cryptographically secure hash function H. If $(R)_x \bmod n = r$, then the signature σ is accepted as valid; otherwise, it is rejected. For the remainder of the paper, unless necessary, we omit the expression $\bmod n$ when writing scalar operations.

ECDSA batch verification: For batch verification, let $\mathcal{S} = \{(r_i, s_i), i = 1, \ldots, t\}$ be a set of t standard ECDSA signatures and let R_i be a point such that $(R_i)_x \bmod n = r_i$, $(R_i)_y = y_i$. Since there are more than one admissible points R_i on the curve for each r_i, the value of y_i that is used for signature generation is unknown to the verifier. This ambiguity is the main reason why well-known methods for batch verification cannot be directly applied to standard ECDSA signatures. In order to verify the signatures in $\mathcal{S} = \{\sigma_i = (r_i, s_i)\}$ as a batch, using NMVR-rand one needs to multiply each verification equation $R_i = u_i B + v_i A_i$ by a sufficiently large randomizer z_i and sum up all the randomized equations:

$$\sum_{i=1}^{t} z_i R_i = \left(\sum_{i=1}^{t} z_i u_i\right) B + \sum_{i=1}^{t} (z_i v_i) A_i, \tag{2}$$

The right-hand side of Equation (2), can be obtained using a multi-scalar multiplication (MSM) algorithm. To verify the validity of the batch of signatures, one checks whether the x-coordinates of both sides of Equation (2) are equal. However, calculating the x-coordinate of $\sum z_i R_i$ by using only r_i values remains the main bottleneck in batch verification of ECDSA.

3 Batch verification of modified ECDSA

Widely used systems, such as Bitcoin and OpenSSL, do not include batch verification support for ECDSA, due to the aforementioned inefficiencies arising from the unknown y-coordinate of the point R. In this work, we investigate batch verification of modified variants of ECDSA.

3.1 Modified ECDSA

The modified variants of standard ECDSA, introduced in [1], do not suffer from the bottleneck caused by missing y-coordinates and shown to provide security

equivalent to that of the original scheme. One such variant is recoverable ECDSA, which we refer to as ECDSA_rec. We use the notation $\sigma_{rec} = (r, s, \psi)$ to denote the signatures produced by ECDSA_rec, where $\sigma = (r, s)$ is a standard ECDSA signature and ψ is a small piece of extra information to recover the point R corresponding to the signature to be verified.

Another modified variant of ECDSA is ECDSA_ast. We denote an ECDSA_ast signature by $\sigma_{ast} = (R, s)$, where R is the randomly chosen point during signature generation, and the component s retains the same definition as in ECDSA. Although the signature size of ECDSA_ast is 50% larger than that of ECDSA, it eliminates the need for point recovery, thereby enabling more efficient batch verification.

3.2 Randomization of Modified ECDSA Verification Equations

As demonstrated in [2], using randomization is essential for batch verification, as omitting randomization allows forgery attacks. Here we consider NMVR-rand [16] and HSS-rand [7] for randomization of verification equations for a batch of modified ECDSA signatures.

Modified versions of ECDSA provide users with the point R instead of the x-coordinate of R, allowing verifiers to use more efficient techniques for batch verification. While ECDSA_ast signatures explicitly include R, ECDSA_rec signatures require a square root computation to recover R using an additional side information ψ that comes with the signature. For the sake of a brief explanation, we assume that we have R in our possession. Therefore, the NMVR-rand based batch verification equation in (2) can be equivalently rewritten as follows:

$$\underbrace{\left(\sum_{i=1}^{t} z_i u_i\right)}_{\hat{z}} B + \sum_{i=1}^{t} (z_i v_i) A_i - \sum_{i=1}^{t} z_i R_i \stackrel{?}{=} \mathcal{O}. \tag{3}$$

The NMVR-rand method chooses z_i values uniformly at random such that the bit length of z_i is half that of the order, i.e., $\text{len}(n)/2$.

On the other hand, HSS-rand samples a random integer U and derives the randomizers z_i using a division-free version of half-way EEA, which is referred to as hEEA_approx_q (Algorithm 1). Given two full-size scalars a and b with $a < b$, hEEA_approx_q produces two half-size scalars c and d satisfying $ca \equiv d \bmod b$. For our application, we choose U as a half-size scalar. For each verification equation, the HSS-rand approach generates two half-size scalars c_i and d_i such that $c_i v_i \equiv U d_i \bmod n$ via hEEA_approx_q with inputs n and v_i/U. By using c_i as the randomizer z_i in (3), we obtain:

$$\underbrace{\left(\sum_{i=1}^{t} c_i u_i\right)}_{\hat{c}} B + U \underbrace{\left(\sum_{i=1}^{t} d_i A_i\right)}_{A} - \sum_{i=1}^{t} c_i R_i \stackrel{?}{=} \mathcal{O}, \tag{4}$$

Algorithm 1 hEEA_approx_q(a, b) [7]

Input: Two integers a and b such that $0 < a < b$
Output: Two integers (c, d) such that $d \equiv ca \bmod b$

$r_0 \leftarrow b$, $r_1 \leftarrow a$, $t_0 \leftarrow 0$, $t_1 \leftarrow 1$, $stop_value \approx \frac{\text{len}(b)}{2}$, $lenr_0 \leftarrow \text{len}(b)$, $lenr_1 \leftarrow \text{len}(a)$
while $lenr_1 > stop_value$ **do**
  $e \leftarrow lenr_0 - lenr_1$
  if $(\text{sign}(r_0) = \text{sign}(r_1))$ **then**
    $r \leftarrow r_0 - (r_1 \ll e)$, $t \leftarrow t_0 - (t_1 \ll e)$
  else
    $r \leftarrow r_0 + (r_1 \ll e)$, $t \leftarrow t_0 + (t_1 \ll e)$
  end if
  if $\text{len}(r) > lenr_1$ **then**
    $r_0 \leftarrow r$, $t_0 \leftarrow t$, $lenr_0 \leftarrow \text{len}(r)$
  else
    $r_0 \leftarrow r_1$, $r_1 \leftarrow r$, $t_0 \leftarrow t_1$, $t_1 \leftarrow t$
    $lenr_0 \leftarrow lenr_1$, $lenr_1 \leftarrow \text{len}(r)$
  end if
end while
return $d = r_1, c = t_1$

where c_i and d_i are approximately $\text{len}(n)/2$ bits long. Computing the left-hand side of Equation (4), requires performing the multi-scalar multiplications given in Equations (5) and (6).

$$A = \left(\sum_{i=1}^{t} d_i A_i \right), \tag{5}$$

$$\hat{c}B + UA - \left(\sum_{i=1}^{t} c_i R_i \right) \stackrel{?}{=} \mathcal{O}. \tag{6}$$

Equations (3) and (4), representing NMVR-rand and HSS-rand, respectively, differ in both the lengths of the scalars involved and the number of MSMs required. The relative performance of the NMVR-rand and HSS-rand methods depends on the value of t as well as on the specific MSM algorithm employed.

3.3 Effect of Endomorphism

Some of the elliptic curves support an efficient endomorphism that enables fast scalar multiplication [8, 9, 14]. Although multidimensional endomorphisms have been proposed in the literature, for the purposes of this paper we restrict our attention to the two-dimensional (2-D) setting. In this case, for a given EC there exists a scalar α that allows αP to be computed efficiently (e.g., with just a single field multiplication), and any $k \in [0, n)$ can be decomposed as $k = k_0 + k_1\alpha$, where k_0 and k_1 are half-size scalars. Then for kP, denoting $P' = \alpha P$, we have the following:

$$kP = k_0 P + k_1(\alpha P) = k_0 P + k_1 P'.$$

In other words, a point multiplication with a full-size scalar is represented as a double scalar multiplication (DSM) with two half-size scalars to reduce the computational cost.

For curves that admit a 2-D endomorphism, the left-hand side of Equation (3) of NMVR-rand batch verification can be computed by splitting the full-size coefficients $z_i v_i$ of variable points A_i into $\hat{v}_{i0}$ and $\hat{v}_{i1}$ such that $z_i v_i = \hat{v}_{i0} + \alpha \hat{v}_{i1}$. In contrast, the two components of HSS-rand batch verification, Equations (5) and (6), require no such transformation for U, c_i and d_i, as their coefficients are already half-size scalars.

Together with this, the term $\hat{z}B$ of the MSM in (3), and similarly $\hat{c}B$ in (6), can be handled in several ways. For $\hat{z}B$, one can use the endomorphism and represent $\hat{z}B = \hat{z}_0 B + \hat{z}_1 B'$ where $\hat{z} = \hat{z}_0 + \alpha \hat{z}_1$ and $B' = \alpha B$ for the endomorphism parameter α. Alternatively, $\hat{z}$ can be split into its upper and lower halves $\bar{z}_0$ and $\bar{z}_1$ such that $\hat{z} = \bar{z}_0 + 2^{\lceil \mathrm{len}(n)/2 \rceil} \bar{z}_1$, which leads to $\hat{z}B = \bar{z}_0 B + \bar{z}_1 \bar{B}$ where $\bar{B} = 2^{\lceil \mathrm{len}(n)/2 \rceil} B$. Another approach is to use a precomputation table for multiples of the fixed base point B. If such a table is available, it can be used directly, and $\hat{z}B$ can be computed independently.

4 Randomness Predictability

In this section, we examine and compare the effects of randomness predictability on the two randomization techniques presented in Section 3.2. In particular, we analyze the number of signatures and conditions under which an attacker can forge them when the verifier's randomness is known or predictable before signing the messages.

4.1 HSS-rand

Recall that the individual verification equation (1) of a modified ECDSA signature $\sigma = (R = (r, y), s)$ on a message m with a verification key A can be expressed as

$$R \stackrel{?}{=} \tfrac{h}{s} B + \tfrac{r}{s} A \tag{7}$$

where $h = H(m)$. Thus, the batch verification equation (4) using HSS-rand with U for t signatures can be rewritten as

$$\left(\sum_{i=1}^{t} c_i \tfrac{h_i}{s_i} \right) B + \sum_{i=1}^{t} c_i \tfrac{r_i}{s_i} A_i - \sum_{i=1}^{t} c_i R_i \stackrel{?}{=} \mathcal{O} \tag{8}$$

where c_i is the half-size scalar randomizer derived by the verifier by calling $\mathsf{hEEA_approx_q}(\frac{r_i}{s_i} \cdot \frac{1}{U}, n)$.

For simplicity, we consider a batch of two signatures, where (8) is expressed as

$$\begin{aligned} \mathcal{O} \stackrel{?}{=}\ & c_1 \tfrac{h_1}{s_1} B + c_1 \tfrac{r_1}{s_1} A_1 - c_1 R_1 \\ & + c_2 \tfrac{h_2}{s_2} B + c_2 \tfrac{r_2}{s_2} A_2 - c_2 R_2 \end{aligned} \tag{9}$$

Attack. Suppose that an attacker controls A_1 and knows the verifier's U (refer to Sec. 3.2). The attacker wants to forge a signature for a message m_2 on behalf of an honest signer who owns A_2. The attacker performs the following steps:

1. Compute $h_1 = H(m_1)$ for a random m_1 and $h_2 = H(m_2)$
2. Select a random $s_2 < n$
3. Compute $R_2 = \frac{h_2}{s_2}B$, and $r_2 = (R_2)_x \bmod n$
4. Call $\mathsf{hEEA_approx_q}(\frac{r_2}{s_2} \cdot \frac{1}{U}, n)$ and obtain c_2
5. Compute $R_1 = c_2\frac{r_2}{s_2}A_2$, and $r_1 = (R_1)_x \bmod n$
6. Select a random $\delta < \sqrt{n}$ s.t. $\text{len}(\delta) < \textit{stop_value}(\approx \frac{\text{len}(n)}{2}$ as in Alg.1) of $\mathsf{hEEA_approx_q}$.
7. Compute $s_1 = r_1 \cdot \frac{1}{U} \cdot \frac{1}{\delta}$
8. Set $A_1 = -\frac{h_1}{r_1}B$

Finally, the attacker sends the modified ECDSA signatures $\sigma_i = (R_i = (r_i, y_i), s_i)$ on messages m_i with verification keys A_i for $i \in \{1, 2\}$.

Correctness of the attack. It relies on a specific behavior of the verifier's process. When the verifier calls the *deterministic* $\mathsf{hEEA_approx_q}$ (i.e., Alg. 1) for the first signature with $\frac{r_1}{s_1} \cdot \frac{1}{U}$, the output is $d_1 = \delta$ and, more importantly, $c_1 = 1$ given that

$$r_1 \cdot \left(\frac{1}{s_1}\right) \cdot \frac{1}{U} = r_1 \cdot \left(\frac{1}{r_1} \cdot U \cdot \delta\right) \cdot \frac{1}{U} = \delta$$

In other words, the verifier effectively calls $\mathsf{hEEA_approx_q}(\delta, n)$. Since the attacker selected δ such that $\text{len}(\delta) < \textit{stop_value}$ of $\mathsf{hEEA_approx_q}$, the algorithm terminates before its first iteration, returning $d_1 = \delta$ and $c_1 = 1$.

Given that $c_1 = 1$, substituting the values in (9) yields

$$\begin{aligned}\mathcal{O} = 1 \cdot \tfrac{h_1}{s_1}B + 1 \cdot \tfrac{r_1}{s_1}(-\tfrac{h_1}{r_1}B) - 1 \cdot (c_2\tfrac{r_2}{s_2}A_2) \\ + c_2\tfrac{h_2}{s_2}B + c_2\tfrac{r_2}{s_2}A_2 - c_2(\tfrac{h_2}{s_2}B),\end{aligned}$$

where the equality follows directly from pairwise cancellation of terms.

Generalization of the attack. The previous attack for a batch size of two can be generalized to any $k \leq t$ signatures within the same batch verification. For each signature i the attacker repeats Steps 1–4. In Step 5 the attacker computes $R_1 = \sum_{i=1}^{k} c_i\frac{r_i}{s_i}A_i$ The remaining Steps 6–8 are the same as in the original attack.

4.2 NMVR-rand

As mentioned earlier, in this approach the verifier selects an individual randomizer z_i for each signature in the batch verification equation (3). For simplicity, we consider a batch of two signatures, where Equation (3) can be expressed as

$$\begin{aligned}\mathcal{O} \stackrel{?}{=} z_1\tfrac{h_1}{s_1}B + z_1\tfrac{r_1}{s_1}A_1 - z_1R_1 \\ + z_2\tfrac{h_2}{s_2}B + z_2\tfrac{r_2}{s_2}A_2 - z_2R_2\end{aligned} \tag{10}$$

Attack. Suppose that an attacker controls A_1 and knows the verifier's randomizers z_1 and z_2. The attacker wants to forge a signature for a message m_2 on behalf of an honest signer who owns A_2. The attacker performs the following steps:

1. Compute $h_1 = H(m_1)$ for a random m_1 and $h_2 = H(m_2)$
2. Select a random $s_2 < n$
3. Compute $R_2 = \frac{h_2}{s_2}B$, and $r_2 = (R_2)_x \bmod n$
4. Compute $R_1 = \frac{1}{z_1} \cdot z_2 \frac{r_2}{s_2} A_2$, and $r_1 = (R_1)_x \bmod n$
5. Set $A_1 = -\frac{h_1}{r_1}B$

Finally, the attacker sends the modified ECDSA signatures $\sigma_i = (R_i = (r_i, y_i), s_i)$ on messages m_i with verification keys A_i for $i \in \{1, 2\}$.

Correctness of the attack. Substituting the values in (10) yields

$$\begin{aligned}\mathcal{O} = z_1 \cdot \tfrac{h_1}{s_1}B + z_1 \cdot \tfrac{r_1}{s_1}(-\tfrac{h_1}{r_1}B) - z_1 \cdot (\tfrac{1}{z_1} \cdot z_2 \tfrac{r_2}{s_2} A_2) \\ + z_2 \tfrac{h_2}{s_2}B + z_2 \tfrac{r_2}{s_2} A_2 - z_2(\tfrac{h_2}{s_2}B)\end{aligned}$$

where the equality follows directly from pairwise cancellation of terms.

Generalization of the attack. Similarly, the previous attack for a batch size of two can be generalized to any $k \leq t$ signatures within the same batch verification, provided that the attacker knows the individual randomizers z_i. For each signature, the attacker repeats Steps 1–3 and then computes $R_1 = \frac{1}{z_1} \sum_{i=1}^{k} z_i \frac{r_i}{s_i} A_i$ in Step 4. The remaining Step 5 is identical to that of the original attack.

Remarks: Although randomness leakage enables the attacker to forge signatures in both randomization techniques, there are key differences. In HSS-rand, leakage of the value of U alone allows the attacker to forge as many signatures as the batch size. In contrast, under NMVR-rand, the attacker must know $k + 1$ individual values of z_i to forge k signatures.

5 Bitcoin and OpenSSL code based Implementation

For our implementation, we adopt the standard ECDSA codes from the Bitcoin [4] and OpenSSL [18] repositories. The Bitcoin implementation uses the curve secp256k1. On the other hand, the OpenSSL ECDSA implementation supports multiple elliptic curves. For our case study, we select secp256r1.

Since the ECDSA_rec implementation is already included in the Bitcoin repository, we only implement ECDSA_ast on secp256k1. On the other hand, since the OpenSSL repository does not provide an implementation for either scheme, we implement both ECDSA_rec and ECDSA_ast on secp256r1, linking them against OpenSSL version 3.5.0.

5.1 Some implementation details

Recovery of R: For the batch verification of ECDSA_rec signatures to succeed, recovering the correct point R from r is essential. According to [1], $\lceil \log_2(h+1) \rceil + 1$ bits of recovery information suffice to correctly recover R, for an EC group with cofactor h. For the curves secp256k1 and secp256r1 used in this work both have cofactor $h = 1$, so 2-bit-long ψ would suffice to recover the point R. Two bits of information implies four points on the curve, particularly $R = (r, y), -R = (r, -y), R' = (r + n, y'), -R' = (r + n, -y')$. As in Bitcoin code, we set the second bit of ψ indicates whether the x-coordinate exceeds n, which occurs with approximate probabilities 2^{-128} and 2^{-130} for secp256k1 and secp256r1, respectively. Similarly, we set the first bit of ψ indicates the sign of the y-coordinate, which may encode either its parity (even or odd) or its sign (positive or negative), depending on the representation used. Therefore, once y^2 is derived by substituting the correct x-coordinate into the curve equation, the parity of the original y-coordinate can be used to select the correct square root, thereby recovering the full point R.

MSM algorithms: For MSM, Bitcoin uses WNAF up to batch size of 87 and then switches to Pippenger. OpenSSL on the other hand uses the Fixed-Windowed MSM method. Our implementation leverages native code of these algorithms provided by Bitcoin and OpenSSL implementations, with minor adjustments when needed to highlight the impact of half-size scalars. For example, we modified the original MSM code to avoid unnecessary endomorphism splitting when the coefficients are already half-size scalars.

Endomorphism in Bitcoin's secp256k1: Since the secp256k1 curve admits an endomorphism, the Bitcoin implementation of ECDSA splits the coefficients of the variable points involved in the MSM, as explained in Section 3.3. It also splits $\hat{z}$ (similarly $\hat{c}$) into its lower and upper halves such that $\hat{z} = \bar{z}_0 + 2^{128}\bar{z}_1$ and benefit from the precomputation tables of the odd multiples of B and $\bar{B} = 2^{128}B$, where B is the generator of the EC group. Therefore, Equation (3) for NMVR-rand is transformed into

$$\bar{z}_0 B + \bar{z}_1 \bar{B} + \sum_{i=1}^{t} \hat{v}_{i0} A_i + \sum_{i=1}^{t} \hat{v}_{i1} A'_i - \sum_{i=1}^{t} z_i R_i \stackrel{?}{=} \mathcal{O} \tag{11}$$

for $A' = \alpha A$, where α is the endomorphism parameter for secp256k1 curve. Consequently, for modified ECDSA batch verification on secp256k1, NMVR-rand requires an MSM involving $3t + 2$ points, whereas HSS-rand requires two MSMs with t and $t + 3$ points, respectively, all involving half-size scalars.

Precomputation in OpenSSL: In OpenSSL implementation of ECDSA on curve secp256r1 [18], the terms $\hat{z}B$ and $\hat{c}B$ and the rest of Equations (3) and (6) are computed separately. For $\hat{z}B$ or $\hat{c}B$, OpenSSL benefits from a large precomputation table (approximately 150KB) for multiples of B. The table contains all

precomputed values $(i\gamma^j)B$ for $i = 1, \ldots, 64$ and $j = 0, \ldots, 36$ and $\gamma = 2^w$ for the window size $w = 7$. Therefore, no point doubling (PD) is required for $\hat{z}B$ or $\hat{c}B$, just 36 point additions (PA) suffice. The remaining part of Equation (3) is computed via the fixed-window scalar multiplication with window size $w = 5$ and is then added to $\hat{z}B$ or $\hat{c}B$. Therefore, for batch verification on the secp256r1 curve, NMVR-rand requires an MSM involving $2t + 1$ points with a mixture of full- and half-size scalars for the variable points, whereas HSS-rand requires two MSMs with t and $t + 2$ points with half-size scalars.

Table 1. Per-signature CC counts for batch verification on secp256k1

	t	HSS-rand	NMVR-rand	Ind.	Impr.[1]	Impr.[2]
ECDSA_rec	2	178749	148510	**127218**	-20.36	-40.51
	3	153045	137694	**127480**	-11.15	-20.05
	4	139799	131695	**126932**	-6.15	-10.14
	5	132354	128573	**126995**	-2.94	-4.22
	6	127167	**126408**	126996	-0.60	-0.13
	8	**120597**	123514	126913	2.36	4.98
	16	**110463**	119021	127048	7.19	13.05
	32	**105310**	116718	127225	9.77	17.22
	64	**102726**	107442	127378	4.39	19.35
	128	**94027**	99069	127413	5.09	26.20
	256	**87106**	91772	127462	5.08	31.66
	512	**81822**	86476	127432	5.38	35.79
ECDSA_ast	2	161311	130590	**127268**	-23.52	-26.75
	3	135252	**119146**	127122	-13.52	-6.40
	4	122636	**113791**	127130	-7.77	3.53
	5	114918	**110539**	127074	-3.96	9.57
	6	109917	**108285**	127104	-1.51	13.52
	8	**103390**	105507	127105	2.01	18.66
	16	**92981**	100653	127134	7.62	26.86
	32	**88056**	98530	127359	10.63	30.86
	64	**85494**	89394	127490	4.36	32.94
	128	**76867**	81007	127481	5.11	39.70
	256	**69934**	73777	127567	5.21	45.18
	512	**64609**	68446	127575	5.61	49.36

[1] Improvement of HSS-rand over NMVR-rand in %.
[2] Improvement of HSS-rand over individual verification in %.

5.2 Experimental platform and Benchmark results

In this section, we describe the details of our implementation and present comparative benchmarking results for batch verification of ECDSA_rec and ECDSA_ast signatures based on our x86-64 implementation. Experimental results were obtained on a system equipped with an Intel Core i7-1360P processor, using a

Table 2. Per-signature CC counts for batch verification on secp256r1

	t	HSS-rand	NMVR-rand	Ind.	Impr. [1]	Impr.[2]
ECDSA_rec	2	279513	296846	**273266**	5.84	-2.29
	3	**240100**	267605	273026	10.28	12.06
	4	**220428**	253162	272994	12.93	19.26
	5	**208538**	244178	272974	14.60	23.61
	6	**200453**	238563	273004	15.97	26.58
	8	**190675**	231125	272806	17.50	30.11
	16	**175774**	220885	272777	20.42	35.56
	32	**168769**	215445	272684	21.66	38.11
	64	**165117**	212509	272625	22.30	39.43
	128	**163238**	211044	272546	22.65	40.11
	256	**162391**	210493	272562	22.85	40.42
	512	**162307**	210381	272585	22.85	40.46
ECDSA_ast	2	**239833**	256698	273007	6.57	12.15
	3	**201578**	228569	272829	11.81	26.12
	4	**182422**	214463	272722	14.94	33.11
	5	**170813**	205964	272788	17.07	37.38
	6	**163135**	200355	272704	18.58	40.18
	8	**153659**	193370	272618	20.54	43.64
	16	**139154**	187735	272625	24.26	48.96
	32	**132364**	178523	272643	25.86	51.45
	64	**128769**	175783	272449	26.75	52.74
	128	**126875**	174388	272466	27.25	53.43
	256	**126023**	173889	272538	27.53	53.76
	512	**125850**	173797	272799	27.59	53.87

[1] Improvement of HSS-rand over NMVR-rand in %.
[2] Improvement of HSS-rand over individual verification in %.

performance core running at a 2.2 GHz clock frequency with Turbo Boost disabled. All implementations were compiled with GCC 11.4.0. For all experiments, 10,000 random instances were generated. Specifically, for batch verification with t signatures, 10,000 batch instances were created, each consisting of t distinct key–message–signature combinations. Each instance was tested 10 times, and the average cycle count was recorded as the final result. Our code is publicly available at repositories https://github.com/iremkp/ossl-ecdsa.git and https://github.com/iremkp/bitcoin_ecdsa.git

Below, we analyze our implementation along three dimensions: signature schemes (ECDSA_rec and ECDSA_ast), randomization techniques (HSS-rand and NMVR-rand), and elliptic curves (secp256k1 and secp256r1), yielding a total of eight distinct configurations.

Tables 1 and 2 present the results for the curves secp256k1 and secp256r1, respectively. Both tables show benchmark results for batch verification of modified ECDSA signatures using the NMVR-rand and HSS-rand methods for different batch sizes. The results are reported in clock cycles (CC) per signature, rounded to the nearest integer. Under columns titled "Ind.", Tables 1 and 2 also report

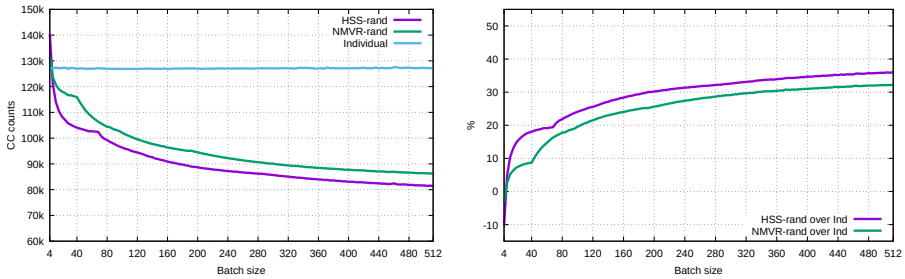


Fig. 1. Per-signature CC counts and % improvements of batch verification for ECDSA_rec on secp256k1

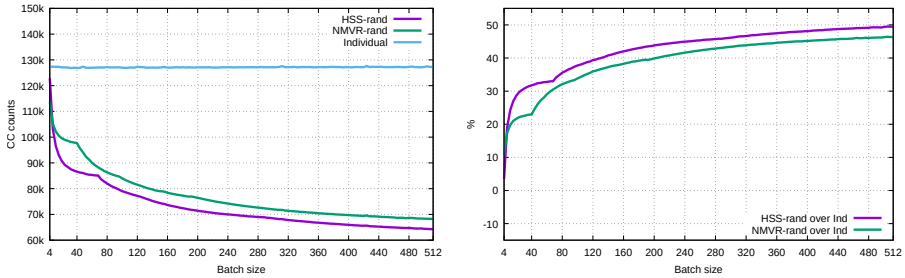


Fig. 2. Per-signature CC counts and % improvements of batch verification for ECDSA_ast on secp256k1

the CC counts for individual verification of standard ECDSA signatures for comparison.

For the secp256k1 curve, HSS-rand batch verification begins to show improvements at a batch size of six for ECDSA_rec and at a batch size of four for ECDSA_ast compared to individual verification as reported in Table 1. The improvement of HSS-rand over NMVR-rand becomes noticeable from a batch size of seven for ECDSA_rec and four for ECDSA_ast. For the secp256r1, as the results in Table 2 indicate, batch verification with HSS-rand begins to exhibit increasing performance improvements at smaller batch sizes than on secp256k1. HSS-rand consistently outperforms NMVR-rand for both ECDSA_rec and ECDSA_ast. Moreover, it always improves over individual verification for ECDSA_ast, while for ECDSA_rec the improvement becomes apparent from a batch size of three.

To visualize batch verification trends across all batch sizes up to 512, including those not listed in the aforementioned tables, we also present the benchmark results as graphs in Figures 1–4. The left-hand graphs in Figures 1 and 2 illustrate the average per-signature clock cycle counts for individual verification of ECDSA signatures (in blue) and for NMVR-rand and HSS-rand batch verification (in green and purple, respectively) of ECDSA_rec and ECDSA_ast signatures on secp256k1. Similarly, Figures 3 and 4 present the corresponding results for the secp256r1 curve. The results are obtained from 100 batch instances contain-

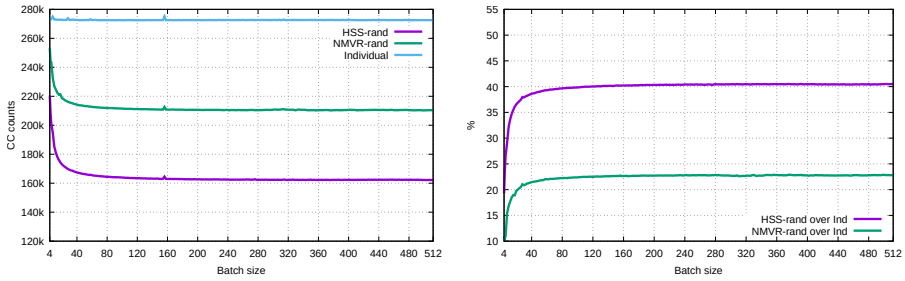


Fig. 3. Per-signature CC counts and % improvements of batch verification for ECDSA_rec on secp256r1

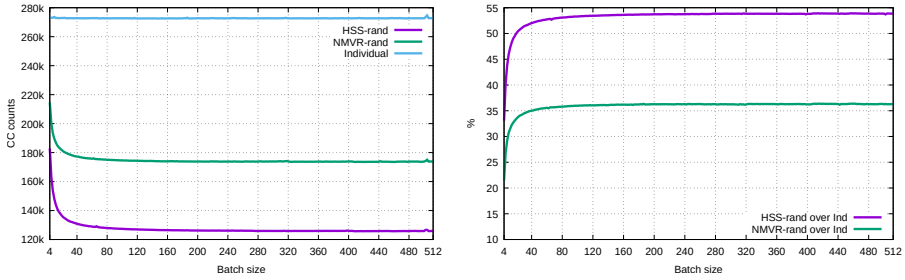


Fig. 4. Per-signature CC counts and % improvements of batch verification for ECDSA_ast on secp256r1

ing t signatures generated with random inputs. For each instance, the algorithm is executed 10 times, and the average clock cycle count is reported. The right-hand graphs in these figures show the corresponding percentage improvements of HSS-rand and NMVR-rand over individual verification. We note that the original Bitcoin implementation recommends a threshold of 88 full-size scalar–point pairs for switching the MSM algorithm from WNAF [21] to Pippenger [19]. Rather than using the scalar multiplication algorithms from the original code verbatim, we introduce targeted modifications to reflect the effect of half-size scalars in batch verification techniques. We conducted independent experiments to determine the optimal switching thresholds for the MSM algorithms used in NMVR-rand and HSS-rand. Our results indicate threshold values of approximately $t \approx 42$ for NMVR-rand and $t \approx 70$ for HSS-rand.

Batch verification of 512 ECDSA_ast signatures using the HSS-rand method on secp256k1 and secp256r1 achieves improvements of approximately 49% and 54%, respectively, over individual verification of ECDSA signatures. For verification of ECDSA_rec signatures with HSS-rand method on secp256k1 and secp256r1, the corresponding improvements at batch size 512 are roughly 35% and 40%, respectively, again over individual verification of standard ECDSA signatures.

5.3 Notes on Individual Verification

Although this paper primarily focuses on batch verification, we also assessed the half-size scalars approach to enhance the individual verification of the modified ECDSA signatures on both curves secp256k1 and secp256r1. Recall the individual verification Equation (1) ($R = uB + vA$). For this, the half-size scalars c and d are derived via hEEA_approx_q with input v. Multiplying Equation (1) by c yields

$$cuB + dA - cR \stackrel{?}{=} \mathcal{O}. \tag{12}$$

In the case of ECDSA_rec on both curves, individual verification does not yield a significant performance improvement, as the benefit gained from using half-size scalars is effectively negated by the computational overhead of recovering the point R from the value r using the recovery information ψ.

Similarly, applying the half-size scalar technique to individual verification of ECDSA_ast signatures on secp256k1 does not yield any improvement. This is because the individual verification in the Bitcoin implementation transforms Equation (1) into the following:

$$R \stackrel{?}{=} \bar{u}_0 B + \bar{u}_1 \bar{B} + \hat{v}_0 A + \hat{v}_1 A', \tag{13}$$

which inherently involves half-size scalars. Consequently, generating half-size scalars with hEEA_approx_q for ECDSA_ast on secp256k1 provides no additional benefit.

In contrast, individual verification of ECDSA_ast signatures using Equation (12) is faster than the verification of standard ECDSA signatures implemented in OpenSSL. The improvements are approximately 18% and 11% with and without a precomputation table for the multiples of B, respectively.

6 Adapting Bos-Coster Algorithm

The Bos–Coster algorithm [5, 6] has been used for MSM to provide improved performance for batch verification of Schnorr-like signatures, and its lower memory footprint compared to windowed methods makes it attractive for certain application scenarios [2, 3, 7]. Having seen the performance improvement offered by HSS-rand in the previous sections, we now investigate the suitability of the Bos–Coster algorithm as an MSM for HSS-rand based batch verification of modified ECDSA. We also implement the Bos–Coster algorithm and adapt it to benchmark batch verification. To the best of our knowledge, neither the OpenSSL nor the Bitcoin repositories previously included a Bos–Coster implementation.

To compute

$$\sum_{i=1}^{t} k_i P_i, \tag{14}$$

the Bos–Coster algorithm operates on two scalar–point pairs at each iteration. It picks the pairs associated with the two largest scalars k_{j_1} and k_{j_2} ($k_{j_1} \geq k_{j_2}$)

during that iteration, and updates the pairs as $(k_{j_1}, P_{j_1}) \leftarrow (k_{j_1} - k_{j_2}, P_{j_1})$ and $(k_{j_2}, P_{j_2}) \leftarrow (k_{j_2}, P_{j_1} + P_{j_2})$. The procedure continues until a single pair (k, P) remains, and it performs one scalar multiplication to return kP as the result of (14). All scalars k_i are assumed to be at most 128 bits throughout this section.

6.1 PA counts for MSM algorithms

For an initial performance comparison, we analyze the number of point additions (PAs) required by each of the following MSM algorithms to compute (14). As mentioned earlier, Fixed-Windowed is used in OpenSSL, and WNAF and Pippenger in Bitcoin.

Bos-Coster MSM: We developed a Python script to count the number of PAs required by the Bos-Coster method. The script was executed 100 times for each t, and the average number of PAs was recorded. Our results for the number of PAs required for (14) align with the close-fit function $2^{7.29}t/\log_2 t$ reported in [22]. For reproducibility, we adopt this function in our analysis.

Fixed-Windowed MSM: Our analysis considers a modified[4] instance of the fixed-window multiplication (FW) algorithm (ecp_nistz256_points_mul in the original code) configured for half-size scalars. To compute (14), our modified FW requires $(37t + 62)$ PAs, where we treat a point doubling (PD) as roughly half the cost of a PA, consistent with the OpenSSL implementation.

WNAF and Pippenger MSM: While the WNAF and Pippenger's methods used in the original code rely on addition with mixed-coordinates (PA_{mix}), efficiently adapting it to the Bos–Coster algorithm is not feasible due to its heap-based structure. Accordingly, we use Jacobian-coordinate addition as PA, where PA, $\mathrm{PA}_{\mathrm{mix}}$, and PD require 16, 11, and 8 field multiplications, respectively. Under this configuration, the WNAF method requires approximately $(47t/2 + 64)$ PAs, whereas Pippenger's algorithm requires $\frac{11}{16}\left(\lceil 128/w \rceil (t + 2^w)\right)$ PAs to compute (14). Pippenger's algorithm determines the bucket window w based on the number of scalar–point pairs rather than the scalar bit-length. For our analysis, we use the optimal bucket window sizes specified in the original code while counting the number of additions for various batch sizes.

6.2 PA counts for batch verification

To obtain the total number of PAs required for HSS-rand, we apply the aforementioned arithmetic complexity estimates to (5) and (6) and sum the resulting costs. Natively, WNAF and FW implementations use precomputation tables for (6). For our analysis, we consider two cases: (i) with a precomputation table

[4] We refrain from modifying the implementation beyond what is necessary for HSS-rand in order to remain as close as possible to the original code.

Table 3. Number of point additions per signature for different MSM schemes used in HSS-rand

	secp256r1				secp256k1			
t	FW	BC	FW*	BC*	WNAF	WNAF*	Pip	BC
4	114	171	133	176	94	93	209	176
8	94	112	103	114	70	70	136	114
16	84	82	89	83	59	58	94	83
32	79	64	81	65	53	53	76	65
64	77	53	78	53	50	50	56	53
128	75	45	76	45	48	48	44	45
256	75	39	75	39	48	48	40	39
512	74	35	74	35	47	47	32	35

FW: fixed-windowed, BC: Bos-Coster, FW*: FW without precomputation table, BC*: BC without precomputation, WNAF: windowed non-adjacent form, WNAF*: WNAF without precomputation, Pip: Pippenger's algorithm.

and (ii) without precomputation. In case (i) for both OpenSSL and Bitcoin implementations, (6) is considered as (14) with $t+1$ points, with an additional cost of 36 and 51 PAs, respectively. For OpenSSL, in case (ii), $\hat{c}$ is split into its lower and upper halves, $\hat{c}_0$ and $\hat{c}_1$, such that $\hat{c}B = \hat{c}_0 + 2^{128}\hat{c}_1$. Accordingly, $\hat{c}B = \hat{c}_0 B + \hat{c}_1 B'$ where $B' = 2^{128}B$ is assumed to be precomputed. On the other hand, in case (ii) for Bitcoin, we use endomorphism splitting for $\hat{c}B$. Therefore, for case (ii) in both implementations, (6) is modeled as (14) with $t+3$ points. The approximate total number of PAs per signature required for HSS-rand for both implementations is presented in Table 3.

The results in Table 3 indicate that, for the secp256r1 curve, the number of point operations required by Bos-Coster is lower than that of fixed-windowed multiplication for batch sizes greater than or equal to 16. In contrast, for secp256k1, Bos–Coster does not appear to improve HSS-rand for batch sizes below 128. Moreover, since the Bitcoin implementation switches to the Pippenger algorithm for MSMs when the number of points exceeds 87, rather than using WNAF, this does not constitute a practical improvement.

6.3 Implementation

We implemented Bos–Coster for HSS-rand to measure the exact performance gain for both curves. Our implementation platform is the same as described in Sec. 5.2. Since batch verification incurs overhead beyond MSM, the improvements reported in Table 3 do not directly translate into performance gains for HSS-rand with Bos–Coster. However, the results of our implementation are largely consistent with those reported in Table 3. Accordingly, we do not report the performance results of Bos-Coster for secp256k1, as they offer no additional performance improvement. For OpenSSL, HSS-rand with Bos–Coster outperforms HSS-rand with the windowed algorithm by approximately 17% for ECDSA_rec and 21% for ECDSA_ast at batch size 512. The results for secp256r1 (OpenSSL) are presented in Table 4. Figures 5 and 6 present the performance trends of

HSS-rand using the Bos–Coster (BC) and fixed-window (FW) algorithms across all batch sizes up to 512, including those not listed in Table 4.

Table 4. Per-signature CC counts for HSS-rand based batch verification on secp256r1 using various MSM schemes

	t	FW	BC	FW*	BC*	Impr.[1]	Impr. [2]
ECDSA_rec	8	190675	196761	196114	201112	-3.19	-2.55
	16	175774	172669	178464	174703	1.77	2.11
	32	168769	158650	170139	159583	6.00	6.20
	64	165117	149066	165761	149613	9.72	9.74
	128	163238	141964	163523	142303	13.03	12.98
	256	162391	137195	162494	137400	15.52	15.44
	512	162307	134734	162338	134907	16.99	16.90
ECDSA_ast	8	153659	160896	159064	165221	-4.71	-3.87
	16	139154	137234	141909	139129	1.38	1.96
	32	132364	123141	133778	123885	6.97	7.40
	64	128769	113320	129452	113654	12.00	12.20
	128	126875	106105	127198	106283	16.37	16.44
	256	126023	101190	126159	101206	19.70	19.78
	512	125850	98925	125935	98914	21.39	21.46

FW: fixed-windowed, BC: Bos-Coster, *: without precomputation table.
[1] Improvement of HSS-rand BC over HSS-rand FW in %.
[2] Improvement of HSS-rand BC* over HSS-rand FW* in %.

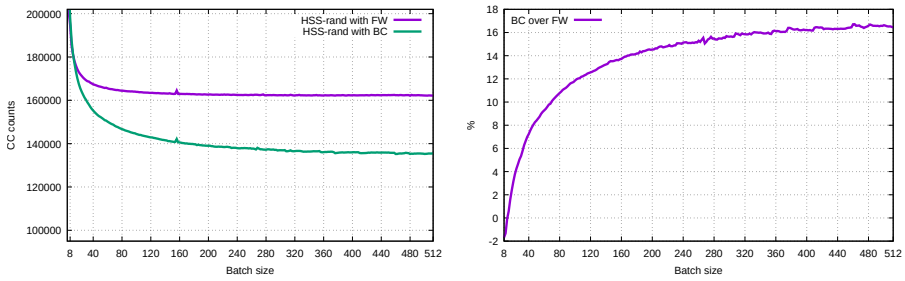


Fig. 5. Per-signature CC counts and % improvements of HSS-rand for ECDSA_rec on secp256r1 using FW and BC

7 Conclusion

In this paper, we have investigated the batch verification of two modified ECDSA schemes, ECDSA_rec and ECDSA_ast, using two randomization techniques, NMVR-rand and HSS-rand. We have also presented the impact of randomizer

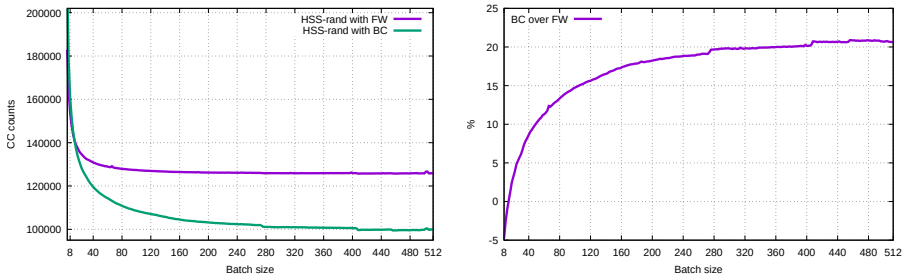


Fig. 6. Per-signature CC counts and % improvements of HSS-rand for ECDSA_ast on secp256r1 using FW and BC

leakage in HSS-rand and NMVR-rand on batch verification, highlighting the critical importance of maintaining randomizer secrecy. Our C implementation, built on the Bitcoin and OpenSSL repositories, demonstrates that HSS-rand outperforms NMVR-rand and individual verification in most cases, especially for larger batch sizes. The results highlight the efficiency gains achievable with HSS-rand in practical batch verification scenarios. Furthermore, we have implemented the Bos–Coster algorithm and demonstrated additional improvements in batch verification of modified ECDSA signatures on the secp256r1 curve. Future work could explore optimizations to further enhance the performance of HSS-rand batch verification on different curves.

References

1. Antipa, A., Brown, D., Gallant, R., Lambert, R., Struik, R., Vanstone, S.: Accelerated verification of ECDSA signatures. In: International Workshop on Selected Areas in Cryptography. pp. 307–318. Springer (2005)
2. Bernstein, D.J., Doumen, J., Lange, T., Oosterwijk, J.J.: Faster batch forgery identification. In: Progress in Cryptology-INDOCRYPT 2012: 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings 13. pp. 454–473. Springer (2012)
3. Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of cryptographic engineering **2**(2), 77–89 (2012)
4. Bitcoin, C.D.: libsecp256k1 (2024), https://github.com/bitcoin-core/secp256k1, gitHub repository
5. Bos, J., Coster, M.: Addition chain heuristics. In: Conference on the Theory and Application of Cryptology. pp. 400–407. Springer (1989)
6. De Rooij, P.: Efficient exponentiation using precomputation and vector addition chains. In: Workshop on the Theory and Application of of Cryptographic Techniques. pp. 389–399. Springer (1994)
7. ElSheikh, M., Paksoy, İ.K., Cenk, M., Hasan, M.A.: Accelerating EdDSA Signature Verification with Faster Scalar Size Halving. IACR Transactions on Cryptographic Hardware and Embedded Systems **2025**(3), 493–515 (2025)
8. Galbraith, S.D., Lin, X., Scott, M.: Endomorphisms for faster elliptic curve cryptography on a large class of curves. Journal of cryptology **24**(3), 446–469 (2011)

9. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Faster point multiplication on elliptic curves with efficient endomorphisms. In: Annual International Cryptology Conference. pp. 190–200. Springer (2001)
10. Josefsson, S., Liusvaara, I.: Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032 (Jan 2017). https://doi.org/10.17487/RFC8032, https://www.rfc-editor.org/info/rfc8032, https://www.rfc-editor.org/info/rfc8032
11. Karati, S., Das, A.: Faster batch verification of standard ECDSA signatures using summation polynomials. In: International Conference on Applied Cryptography and Network Security. pp. 438–456. Springer (2014)
12. Karati, S., Das, A., Roychoudhury, D.: Randomized batch verification of standard ECDSA signatures. In: Security, Privacy, and Applied Cryptography Engineering: 4th International Conference, SPACE 2014, Pune, India, October 18-22, 2014. Proceedings 4. pp. 237–255. Springer (2014)
13. Karati, S., Das, A., Roychowdhury, D., Bellur, B., Bhattacharya, D., Iyer, A.: Batch verification of ECDSA signatures. In: Progress in Cryptology-AFRICACRYPT 2012: 5th International Conference on Cryptology in Africa, Ifrance, Morocco, July 10-12, 2012. Proceedings 5. pp. 1–18. Springer (2012)
14. Longa, P., Sica, F.: Four-dimensional Gallant–Lambert–Vanstone scalar multiplication. Journal of Cryptology **27**(2), 248–283 (2014)
15. Moon, A.: ed25519-donna: Implementations of a Fast Elliptic-Curve Digital Signature Algorithm. https://github.com/floodyberry/ed25519-donna, original-date: 2012-03-02T22:15:52Z, Accessed: 2024-10-09
16. Naccache, D., M'RaÏhi, D., Vaudenay, S., Raphaeli, D.: Can DSA be improved?—Complexity trade-offs with the digital signature standard—. In: Advances in Cryptology—EUROCRYPT'94: Workshop on the Theory and Application of Cryptographic Techniques Perugia, Italy, May 9–12, 1994 Proceedings 13. pp. 77–85. Springer (1995)
17. NIST (National Institute of Standards and Technology): NIST-SP-800-186-Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters. Tech. Rep. SP 800-186, U.S. Department of Commerce (February 2023), https://doi.org/10.6028/NIST.SP.800-186
18. OpenSSL, S.F.: Openssl (2024), https://github.com/openssl/openssl, gitHub repository
19. Pippenger, N.: The minimum number of edges in graphs with prescribed paths. Mathematical Systems Theory **12**(1), 325–346 (1978)
20. Qu, M.: Sec 2: Recommended elliptic curve domain parameters. Certicom Res., Mississauga, ON, Canada, Tech. Rep. SEC2-Ver-0.6 (1999)
21. Straus, E.G.: Addition chains of vectors (problem 5125). American Mathematical Monthly **70**(806-808), 16 (1964)
22. Wesolowski, M.: Batch Verification of Elliptic Curve Digital Signatures. Master's thesis, University of Waterloo (2015)