# A Parallelization of ECDSA Resistant to Simple Power Analysis Attacks

Sarang Aravamuthan[1] and Viswanatha Rao[2]

[1] Advanced Technology Center, Hyderabad, India.
[2] Embedded Systems Group, Bangalore, India.*

E-mail: {a.sarangarajan, viswanatharao.t}@tcs.com

**Abstract.** The Elliptic Curve Digital Signature Algorithm admits a natural parallelization wherein the point multiplication step can be split in two parts and executed in parallel. Further parallelism is achieved by executing a portion of the multiprecision arithmetic operations in parallel with point multiplication. This results in a saving in timing as well as gate count when the two paths are implemented in hardware and software. This article attempts to exploit this parallelism in a typical system context in which a microprocessor is always present though a hardware accelerator is being designed for performance. We discuss some implementation aspects of this design with reference to power analysis attacks. We show how the Montgomery point multiplication and the binary extended gcd algorithms can be adapted to prevent simple power analysis attacks.

We implemented our design using a hardware/software parallel architecture. We present the results when the software component is coded on an 8051 architecture and an ARM7TDMI processor.

**Keywords.** authentication, ECDSA, parallel implementation, power analysis attacks.

## 1 Introduction

Public key cryptosystems emerged in the 1970's with the advent of the Diffie Hellman key exchange method that allowed two parties to negotiate a secret key over a public channel. Since then, RSA has emerged as the prevalent public key cryptosystem due to the accepted hardness of the factorization problem on which its security is based.

In recent years, Elliptic Curve Cryptography (ECC) has emerged as an attractive alternative to RSA. The smaller key size of ECC leads to

* Both ATC and ESG are research units of Tata Consultancy Services Limited.

faster computations making it the cryptosystem of choice on constrained devices such as smart cards where memory and computing power are limited.

The primary objectives of a public key cryptosystem are to enable authentication and key exchange. Digital signatures using, for instance, the *Elliptic Curve Digital Signature Algorithm* (ECDSA) authenticate the signer since the signer's private key is required to generate the signature. Key exchange algorithms such as *Elliptic-curve Diffie Hellman* (ECDH) allow two parties to negotiate a secret key over a public channel.

The computationally expensive step in both ECDSA and ECDH is the point multiplication operation. This operation is of the form $k.G$ where $G$ is referred to as the base point and $k$ is an integer less than the order of $G$. While $G$ is fixed in ECDSA, it varies for each instance of ECDH.

Embedded resource constrained devices such as smart cards benefit from a parallel implementation of these ECC algorithms. Assuming that it is necessary to implement the algorithm in customized hardware logic for performance reasons and noting the fact that every system has a processor, it is advantageous to use any parallelism. The net effect of this would be to improve the performance of the algorithm as well as reduce the gate count requirement for the customized hardware logic.

In this article, we first describe an efficient implementation of ECDSA signature generation using a parallel architecture. We assume the availability of two processing units $\mathcal{P}_1$ and $\mathcal{P}_2$ with $\mathcal{P}_2$ faster than $\mathcal{P}_1$ by a factor of $\Theta \geq 1$. Our implementation splits the algorithm in the following manner

- $\mathcal{P}_1$ executes some of the multiprecision arithmetic steps and a portion of the point multiplication operation.
- $\mathcal{P}_2$ executes the remaining portion of the point multiplication operation.

We discuss how this parallelism can be achieved using a single precomputed point. Our implementation improves the performance of point multiplication by a factor of approximately $(\Theta + 1)$. However, the execution of multiprecision arithmetic steps in $\mathcal{P}_1$ reduces the speedup for the overall digital signature algorithm to less than $(\Theta + 1)$.

Our design has the advantage of requiring no synchronization or sharing of memory between the two processing units. The two units run independently of each other. This is in contrast to other parallel implementations of the point multiplication [8, 12, 22] where each iteration of

the multiplication step is split to execute along parallel paths leading to shared memory.

Power Analysis attacks on cryptographic systems study the power consumed during the cryptographic operation of interest to deduce some secret information such as the user's private key. In Simple Power Analysis (SPA), the attacker directly correlates the power consumed to the instructions performed by the processor. In Differential Power Analysis (DPA), the attacker uses statistical analysis to extract the hidden information from a large sample of power traces.

In the implementation of ECDSA, both point multiplication and the modular inverse operation can be subject to SPA attacks. We describe these attacks for the Montgomery point multiplication and the binary extended gcd algorithms. Following the approach in [12], we show how these operations can be made SPA resistant under the assumption that the power trace of a swap() function of two variables is indistinguishable.

This paper is structured as follows. In the next section, we describe the ECDSA signature generation algorithm. In Section 3, we describe parallelization of this algorithm and discuss some design trade-offs. Sections 4 and 5 describe SPA attacks and their countermeasures for the point multiplication and modular inverse operations. In Section 6, we provide the implementation details and conclude with Section 7.

### 1.1 Related Work

Survey chapters on ECC and side-channel resistant implementations can be found in [2, 4]. The ANSI X9.62 ECDSA standard is described in [13].

The computationally intensive step in ECDSA is point multiplication and most researchers have concentrated on efficient ways of implementing this in hardware or software. A survey of different point multiplication algorithms as well as the underlying field operations in prime and binary fields can be found in [3, 10]. The Montgomery ladder on which the point multiplication algorithm in Section 4 is based is analyzed in [16].

Practical implementations of the point multiplication step as well as the signature algorithm have been carried out by several researchers. An implementation of the point multiplication on a 134-bit field using an 8051 microcontroller is described in [25]. In [11], the authors describe an implementation of ECDSA over a 160-bit prime field on a Mitsubishi

(M16C) microprocessor. The authors report a timing of 150 ms to generate a signature. In [6], a design of a cryptographic processor for arbitrary elliptic curves over GF($2^m$) is described.

A number of papers discuss parallelism of the point multiplication step. In [12], the authors describe variants of the Montgomery method and the fixed base comb method of Lim and Lee [19] that lead to a parallel implementation and is resistant to side channel attacks. In [22], a $2^w$-ary point multiplication method is described that is resistant to side channel attacks and is parallelizable. Other parallel implementations of the point multiplication step are discussed in [8, 9].

Power analysis attacks were introduced by Kocher et. al. [18]. The first work to study these attacks on ECC and suggest countermeasures was the paper by Coron [5]. Randomization techniques to defeat DPA attacks are described in [15]. Special forms of elliptic curves that use the same formulas for add and double operations to thwart SPA attacks are presented in [14].

## 2 ECDSA Signature Generation

The standard signature generation algorithm is detailed below. $n$ is the order of the base point $G = (x_0, y_0)$ of the elliptic curve, $t$ is the message to be signed and $d < n$ is the signer's private key. We assume $n$ to be a large prime number.

**Algorithm 1**: ECDSA Signature Generation

1. Compute the message digest $e = \text{SHA-1}(t)$.
2. Generate a random number $k$, $0 < k < n$.
3. Compute the value $k.G = (x_1, y_1)$. Let $r = x_1 \mod n$. If $r = 0$, go to step 2.
4. Compute $z_0 = (e + dr) \mod n$.
5. Compute $s = k^{-1} z_0 \mod n$. If $s = 0$, go to step 2.
6. Output the signature $(r, s)$.

The message digest is computed using a standard hash algorithm such as SHA-1. The signature is generated on the hash and is a pair of multi-precision integers $(r, s)$.

Step 3 is the point multiplication operation. Step 4 requires a modular multiplication. The modular reduction can be computed using, for example, the method of Barrett [1].

Step 5 requires a modular division operation to compute $s$ so that $ks = z_0 \mod n$. This can be realized by adapting a modular inversion algorithm such as the binary extended gcd to perform division (see Section 5 for details).

## 3 A Parallel Implementation

Our approach to the parallelization of Algorithm 1 hinges on two observations.

- The point multiplication step (step 3) can be parallelized.
- A part of the multiprecision arithmetic computation (steps 4 and 5) is independent of the point multiplication operation.

We assume the availability of two processing units $\mathcal{P}_1$ and $\mathcal{P}_2$ to carry out Algorithm 1. Let $\Theta$ be the ratio of the computing speed of $\mathcal{P}_2$ to that of $\mathcal{P}_1$. We assume $\Theta \geq 1$, i.e. $\mathcal{P}_2$ is at least as fast as $\mathcal{P}_1$. $\Theta$ is a predetermined constant that could be estimated by experimentation.

We split the point multiplication step in the following manner. Let $m$ be the size of $n$ in bits. Since $k < n$, $k$ is an $m$ bit number. Let $m_1 = \lfloor m/(\Theta + 1) \rfloor$ and $m_2 = m - m_1$. Note that $m_2 \approx \Theta m_1$. Let

$$Q = 2^{m_1}.G \tag{1}$$

be precomputed and stored. We split $k$ as

$$k = k_1 + 2^{m_1} k_2$$

where $k_1$ (resp. $k_2$) is the number formed by the $m_1$ least (resp. $m_2$ most) significant bits of $k$. Then, by (1),

$$k.G = k_1.G + 2^{m_1} k_2.G = k_1.G + k_2.Q \tag{2}$$

The idea is for $\mathcal{P}_1$ to compute $k_1.G$ and $\mathcal{P}_2$ to compute $k_2.Q$ and finally add the two points to get $k.G$. Since $k_2$ has $\Theta$ times as many bits as $k_1$, $\mathcal{P}_1$ and $\mathcal{P}_2$ will take approximately the same time to perform their computations.

Next, we split steps 4 and 5 in Algorithm 1 as

(a) Compute $z_1 = k^{-1} \mod n$
(b) Compute $z_2 = z_1 d \mod n$
(c) Compute $z_3 = z_1 e \mod n$

(d) Compute $s = (z_3 + z_2 r) \mod n$

Steps (a), (b) and (c) are independent of the point multiplication step and could be executed by $\mathcal{P}_1$ or $\mathcal{P}_2$ in addition to its share of point multiplication. However, we focus on an embedded system context where a microprocessor always exists and a hardware accelerator is implemented for improving performance of ECDSA. In this context, $\mathcal{P}_1$ is the microprocessor and $\mathcal{P}_2$ is the hardware accelerator. Thus, it is preferable for $\mathcal{P}_1$ to execute these additional steps. This reduces the gate count of $\mathcal{P}_2$.

We observe that the computation of $k.G$ in (2) can also be implemented serially with the same efficiency using the method of joint scalar multiplication of Shamir [7] or Solinas [24]. The authors are not aware of any studies on making these algorithms SPA-proof. This is not surprising as these methods are used more in signature verification. A naive approach to making these methods secure from SPA attacks increases their complexity beyond the SPA-proof Montgomery method (Algorithm 3.1) described in Section 4.

The parallelized version of Algorithm 1 is presented below.

---

**Algorithm 2**: ECDSA Signature Generation: A Parallel Implementation

---

1. Compute the message digest $e = \text{SHA-1}(t)$.
2. Generate a random number $k$, $0 < k < n$.
3. Compute $z_1 = k^{-1} \mod n$ $\|$ Compute $P_2 = k_2.Q$
4. Compute $z_2 = z_1 d \mod n$ $\|$
5. Compute $z_3 = z_1 e \mod n$ $\|$
6. Compute $P_1 = k_1.G$ $\|$
7. Compute $k.G = P_1 + P_2 = (x_1, y_1)$. Let $r = x_1 \mod n$.
8. Compute $s = (z_3 + z_2 r) \mod n$. If $r = 0$ or $s = 0$, go to 2.
9. Output the signature $(r, s)$.

---

The steps marked by $\|$ are computed in parallel; the left side is computed by $\mathcal{P}_1$ and the right side by $\mathcal{P}_2$. The remaining steps are computed by $\mathcal{P}_1$.

### 3.1 Implications of Parallelization

Algorithm 2 provides a more efficient implementation of ECDSA. The benefits of parallelization and some design issues are discussed below.

**A reduction in timing:** We calculate the improvement in timing with respect to a serial implementation of Algorithm 1 on $\mathcal{P}_1$. Let $t_m, t_i, t_a$ and $t_p$ denote the time taken by $\mathcal{P}_1$ to perform a modular multiplication, modular inversion, point addition and point multiplication respectively. The complexity of these operations suggest the following ordering of these values.

$$t_m < t_i \approx t_a \ll t_p$$

Assuming that the cost of modular division and inversion are the same (this assumption is justified in Section 5), the time taken by Algorithm 1 is $t_p + t_m + t_i$.[3]

In Algorithm 2, steps 3 through 6 compute a modular inversion, two modular multiplications and a point multiplication in parallel. Thus the effective cost of these operations is $\frac{t_p+t_i+2t_m}{\Theta+1}$. The cost of steps 7 and 8 is $t_a + t_m$. We calculate the speedup of Algorithm 2 as a ratio of these timings

$$\frac{t_p + t_m + t_i}{((t_p + t_i + 2t_m)/(\Theta + 1) + t_a + t_m)} \approx (\Theta + 1)(1 - \alpha)$$

where $\alpha \approx (\Theta + 1)(t_m + t_a)/t_p$.

**Splitting the point multiplication step:** In (1), $m_1$ was chosen in such a way that the time taken by $\mathcal{P}_1$ and $\mathcal{P}_2$ to compute $k_1.G$ and $k_2.Q$ is approximately the same. However, in Algorithm 2, $\mathcal{P}_1$ also performs multiprecision arithmetic operations in parallel with $\mathcal{P}_2$.

A more precise estimate of $m_1$ follows from equating the time taken by $\mathcal{P}_1$ to perform the parallel operations with the overall time for these operations. This gives

$$\frac{m_1 t_p}{m} + 2t_m + t_i = \frac{t_p + t_i + 2t_m}{\Theta + 1}$$

which yields

$$m_1 = \frac{m}{\Theta + 1} \left\{ \frac{t_p - \Theta(t_i + 2t_m)}{t_p} \right\} \tag{3}$$

[3] We ignore the cost of the Steps 1 and 2. $e$ is assumed to be given as input to the algorithm while the cost of step 2 is assumed to be negligible.

**The case of $\Theta \gg 1$:** If $\Theta \gg 1$, then $m_1$ can be set to 0 and the point multiplication can be performed entirely by $\mathcal{P}_2$. For example, if $\mathcal{P}_1$ is an 8051 microprocessor and $\mathcal{P}_2$ is a dedicated hardware accelerator for performing point multiplication, then in Algorithm 2, $\mathcal{P}_1$ would execute only steps 3, 4 and 5 and skip step 6. The point addition in step 7 would be skipped as well.

**Masking of power trace through parallelization:** SPA attacks in ECDSA attempt to discover the ephemeral key $k$ by measuring the power dissipated during operations involving $k$ such as the point multiplication or the modular inversion step. Using $k$ and the signature, the attacker can determine the secret key $d$ through $d = r^{-1}(ks - e) \mod n$.

Algorithm 2 has the potential for resistance against such attacks. This is because the point multiplication and modular inversion steps are executed in parallel and the power trace of the two paths mask each other.

As masking is not generally perceived as an effective countermeasure against side channel attacks, we describe more robust ways of securing the operations of interest from SPA threats.

## 4 Securing the Point Multiplication Step

An efficient algorithm for point multiplication that requires no precomputation is the Montgomery method. The algorithm applies to curves defined over GF($2^m$) and requires $6m + 10$ field multiplications and 1 field inversion. The algorithm is based on an idea of Montgomery [23] and was developed by López and Dahab [20].

In Montgomery's method, affine points $(x, y) \in \text{GF}(2^m)$ are represented in projective co-ordinates as triples $(X, Y, Z)$ such that $x = X/Z$ and $y = Y/Z$. The algorithm uses the fact that for a fixed point $P = (X, Y, 1)$ and points $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$ with $P_2 = P_1 + P$, the $X$ and $Z$ coordinates of $P_1 + P_2$, $2P_1$ and $2P_2$ can be expressed in terms of the $X$ and $Z$ coordinates of $P_1$, $P_2$ and $P$.

At iteration $j$, the algorithm maintains two points $(l.P, (l + 1).P)$ where $l$ is the value given by the $j$ leftmost bits of $k$. In the next iteration the algorithm computes the pair $(2l.P, (2l+1).P)$ or $((2l+1).P, (2l+2).P)$ depending on whether the $(j+1)$st bit from the left is 0 or 1. The algorithm returns with the $x$ coordinates of $k.P$ and $(k + 1).P$ from which the $y$ coordinate of $k.P$ is derived as shown below.

**Algorithm 3**: Montgomery Point Multiplication

INPUT: $k = (k_{m-1}, \ldots, k_0)_2$ with $k_{m-1} = 1$ and $P = (x, y) \in E(\mathrm{GF}(2^m))$ with curve parameter $b$.
OUTPUT: $k.P$.

1. $X_1 \leftarrow x, Z_1 \leftarrow 1, X_0 \leftarrow x^4 + b, Z_0 \leftarrow x^2$. {Compute $(P, 2P)$}
2. For $i$ from $m-2$ to 0 do
   2.1 If $k_i = 1$ then
   $T \leftarrow Z_1, Z_1 \leftarrow (X_1.Z_0 + X_0.T)^2$, $X_1 \leftarrow x.Z_1 + (X_1 Z_0).(X_0 T)$
   $T \leftarrow X_0$, $X_0 \leftarrow X_0^4 + b.Z_0^4, Z_0 \leftarrow T^2.Z_0^2$
   2.2 Else
   $T \leftarrow Z_0, Z_0 \leftarrow (X_1.T + X_0.Z_1)^2$, $X_0 \leftarrow x.Z_0 + (X_1 T).(X_0 Z_1)$
   $T \leftarrow X_1$, $X_1 \leftarrow X_1^4 + b.Z_1^4, Z_1 \leftarrow T^2.Z_1^2$
3. $x_2 \leftarrow X_1/Z_1$.
4. $y_2 \leftarrow (x+X_1/Z_1)[(X_1+xZ_1)(X_0+xZ_0)+(x^2+y)(Z_1Z_0)](xZ_1Z_0)^{-1}+y$.
5. Return $(x_2, y_2)$.

If the power trace of this algorithm enables an attacker to distinguish between steps 2.1 and 2.2, then she will be able to recover the secret value $k$. While distinguishing between these two steps may be hard, due to the similarity of the operations carried out, nevertheless a more SPA resistant implementation of this algorithm is desirable.

We describe our proposed modifications in the following algorithm.

**Algorithm 3.1**: Our proposed modification (SPA resistant)

INPUT: $k = (k_{m-1}, \ldots, k_0)_2$ with $k_{m-1} = 1$ and $P = (x, y) \in E(\mathrm{GF}(2^m))$.
OUTPUT: $k.P$.

1. $X_1 \leftarrow x, Z_1 \leftarrow 1, X_0 \leftarrow x^4 + b, Z_0 \leftarrow x^2$. {Compute $(P, 2P)$}
2. For $i$ from $m-2$ to 0 do
   $R_1 \leftarrow X_1.Z_0$
   $R_2 \leftarrow X_0.Z_1$
   $R_3 \leftarrow X_{1-k_i}$
   $Z_{k_i} \leftarrow (R_1 + R_2)^2$
   $X_{k_i} \leftarrow x.Z_{k_i} + R_1.R_2$
   $X_{1-k_i} \leftarrow R_3^4 + b.Z_{1-k_i}^4$
   $Z_{1-k_i} \leftarrow R_3^2.Z_{1-k_i}^2$
3. $x_2 \leftarrow X_1/Z_1$.
4. $y_2 \leftarrow (x+X_1/Z_1)[(X_1+xZ_1)(X_0+xZ_0)+(x^2+y)(Z_1Z_0)](xZ_1Z_0)^{-1}+y$.
5. Return $(x_2, y_2)$.

### 4.1 Analysis of Algorithm 3.1

The correctness of Algorithm 3.1 is verified by substituting $k_i = 0$ (resp. $k_i = 1$) and confirming that the variables $X_0, Z_0, X_1$ and $Z_1$ are updated in the same manner as in Algorithm 3.

Assignments such as $R_3 \leftarrow X_{1-k_i}$ are realized through indirect addressing modes. The algorithm maintains the addresses of $X_{k_i}$, $Z_{k_i}$, $X_{1-k_i}$, $Z_{1-k_i}$ in variables $aXk$, $aZk$, $aXk'$, $aZk'$ respectively. Let addr $X$ denote the address of $X$. The variables are initialized as

$aXk \leftarrow$ addr $X_1$
$aZk \leftarrow$ addr $Z_1$
$aXk' \leftarrow$ addr $X_0$
$aZk' \leftarrow$ addr $Z_0$

and updated at the start of each iteration as

if $k_i \neq k_{i+1}$
    swap $(aXk, aXk')$
    swap $(aZk, aZk')$

where swap() is a function to swap two variables. Now, all the operations in step 2 dependent on $k_i$ are realized through indirect addressing modes.

Thus, the only operations in Algorithm 3.1 dependent on $k_i$ are the swap() functions. Assuming that the power trace of this operation is indistinguishable, Algorithm 3.1 is secure against SPA. We formulate this as

**Assertion 1:** *Algorithm 3.1 is secure from SPA on a computing architecture whose power trace of a swap() function is indistinguishable. Moreover, Algorithm 3.1 requires the same number of field multiplications and inversion operations as Algorithm 3.* □

**Security Against Fault Attacks:** In [16], the authors describe how inducing computational faults or memory errors during the execution of the algorithm may lead one to recover bit(s) of the secret key. The first type of attack is known as *C safe-error* attack and the second *M safe-error* attack. Both attacks extract the secret over several iterations, each iteration recovering a single bit of the key. Thus they do not apply to Algorithm 3.1 as the key is ephemeral. Moreover, the C safe-error attack relies on the presence of dummy operations (operations that do not influence the final result) in the algorithm. All the steps in Algorithm 3.1 influence the final result, thus securing it from C safe-error attacks.

## 5 Securing the Modular Inversion Step

In Algorithm 2, the computation of the signature requires a modular inversion operation to evaluate $k^{-1} \mod n$. If not implemented carefully, a power analysis of this operation can reveal the bits of $k$ which can be used to extract the secret key $d$.

Modular inversion is normally achieved through the binary extended gcd algorithm [21, 14.61]. Algorithm 4 maintains the identities $Ak = u \mod n$, $Ck = v \mod n$ and $\gcd(n, k) = \gcd(u, v)$ through each iteration. When the algorithm terminates, $u = 0$ and thus $v = \gcd(n, k) = 1$ and $C = k^{-1} \mod n$.

---

**Algorithm 4**: Binary extended gcd algorithm

---

INPUT: Integers $k < n$ where $n$ is a large prime.
OUTPUT: $C = k^{-1} \mod n$.

1. $u \leftarrow k, v \leftarrow n, A \leftarrow 1, C \leftarrow 0$.
2. While $u$ is even do
   2.1 $u \leftarrow u/2$
   2.2 if $A$ is even, then $A \leftarrow A/2$; else $A \leftarrow (A + n)/2$
3. While $v$ is even do
   3.1 $v \leftarrow v/2$
   3.2 if $C$ is even, then $C \leftarrow C/2$; else $C \leftarrow (C + n)/2$
4. If $u \geq v$ then $u \leftarrow u - v, A \leftarrow A - C$
   else $v \leftarrow v - u, C \leftarrow C - A$.
5. If $u = 0$, then {if $C > 0$ then return$(C)$ else return$(n + C)$};
   else go to step 2.

---

Algorithm 4 can be adapted to perform a modular division [17, 4.5.2]. To compute $k^{-1}z_0 \mod n$, step 1 is modified by replacing $A \leftarrow 1$ by $A \leftarrow z_0$. The algorithm now maintains the identities $Ak = uz_0 \mod n$, $Ck = vz_0 \mod n$ and $\gcd(n, k) = \gcd(u, v)$ through each iteration. As before, the algorithm terminates with $v = 1$ and thus $C = k^{-1}z_0 \mod n$.

In each iteration of Algorithm 4, either step 2 or step 3 is executed but not both. If, in each iteration, an attacker can determine how many times step 2 or 3 is executed as well as which of the two possibilities occurred in step 4, then she can work backwards and recover $k$. To see this, assume that the symbols $U, V, D$ stand for the following operations: $V$ refers to the subtraction step $v \leftarrow v - u$, $U$ to the step $u \leftarrow u - v$ and $D$ to the "divide by 2" operation of $u$ or $v$.

Suppose from the power trace, the attacker extracts the sequence

$$X_1 D^{\alpha_1} X_2 D^{\alpha_2} \dots X_j D^{\alpha_j}$$

where each $X_i$ stands for a $U$ or a $V$ and $D^{\alpha_i}$ is $\alpha_i$ copies of $D$ and $j$ is the number of times step 4 is executed. The attacker now recovers $k$ in the following manner.

---

**Analysis 1**: Recovery of $k$ through SPA attack on Algorithm 4

Set $u \leftarrow 1, v \leftarrow 1$.
For $i$ from $j$ down to 1 do
    if $X_i = U$ then $u \leftarrow 2^{\alpha_i} u + v$
    else $v \leftarrow 2^{\alpha_i} v + u$
Return $k = u$.

---

A simple way to secure this from power analysis is to first multiply $k$ by a random number $\gamma$, compute $(k\gamma)^{-1} \mod n$ and multiply back by $\gamma$ and reduce to recover $k^{-1} \mod n$. However, this requires two additional modular multiplication operations.

Our solution is to render the operations $U$ and $V$ indistinguishable to the attacker. Then Analysis 1 would yield $2^j$ possible values for $k$. As $j = O(\log n)$[‡], the attack is made computationally expensive.

As before, our solution relies on address arithmetic. We store the addresses of variables $u, v, A$ and $C$ in $auv_+, auv_-, aAC_+$ and $aAC_-$ respectively if $u \geq v$ and in $auv_-, auv_+, aAC_-$ and $aAC_+$ otherwise.

Let $\text{sub}(aX, aY)$ stand for the operation $\{*aX \leftarrow *aX - *aY\}$ where $*aX$ is the value pointed to by $aX$. We replace step 4 of Algorithm 4 by the following sequence of operations.

---

**Step $\tilde{4}$**: Replacement for step 4 of Algorithm 4

$auv_+ \leftarrow \text{addr}\, u$
$auv_- \leftarrow \text{addr}\, v$
$aAC_+ \leftarrow \text{addr}\, A$
$aAC_- \leftarrow \text{addr}\, C$
if $(u < v)$
    swap $(auv_+, auv_-)$
    swap $(aAC_+, aAC_-)$

[‡] Algorithm 4 is analyzed in [17]. Empirical tests show that the average value of $j \approx 0.5 \log n + 0.203 \log k$; see [17, pp. 348–352] for details.

sub $(auv_+, auv_-)$
sub $(aAC_+, aAC_-)$

---

The only conditional statements in step $\tilde{4}$ are the swap() operations. Assuming these have indistinguishable power, the modified algorithm is resistant to SPA attacks.

Let **Algorithm 4.1** be Algorithm 4 with step 4 replaced by step $\tilde{4}$. Then we have

**Assertion 2:** *Algorithm 4.1 is secure from SPA on a computing architecture whose power trace of a swap() function is indistinguishable.* □

## 6 Implementation Results

We present the results of applying the parallelism described in Section 3, to a 163-bit binary curve recommended by NIST (see Curve B-163 in [13]).

We have assumed the use of a software platform and a hardware platform for $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. In one study an ARM7TDMI platform was used for software. In another study an 8051 platform was employed for software. In both studies, a Xilinx Virtex-II FPGA was used as the hardware platform. Both hardware and software were run at a 10 MHz clock. The low clock speed was chosen keeping in mind applications such as smart cards.

The following subsections provide the results for the two studies.

### 6.1 ARM7TDMI as $\mathcal{P}_1$ and Xilinx Virtex-II FPGA as $\mathcal{P}_2$

As per Algorithm 2, the multiprecision arithmetic was run on the software platform and the point multiplication operation was parallelized to execute on both platforms using Algorithm 3.1. It was noted through implementation that the hardware platform performance was 3.1 times faster than the software platform for point multiplication. The point multiplication on software was carried out independently and timed 324 ms. Using $\Theta = 3.1, t_p = 324$ and the values for $t_i$ and $t_m$ from Table 1, Equation 3 yields $m_1 \approx 34$. Thus, the partition used in this study was 129 bits for hardware and 34 bits for software. The results are tabulated below.
With the above results, the total time for generation of a signature was 94.02 ms.

| Algorithm | Time (ms) |
|---|---|
| Modular multiplication on $\mathcal{P}_1$ | 1.82 |
| Modular inversion on $\mathcal{P}_1$ | 11.3 |
| Point multiplication, 34 bits, on $\mathcal{P}_1$ | 72.75 |
| Point multiplication, 129 bits, on $\mathcal{P}_2$ | 89 |
| Point addition on $\mathcal{P}_1$ | 3.2 |

**Table 1.** Implementation Statistics for ECDSA on ARM7TDMI and Xilinx Virtex-II

The clock could be increased for both the hardware and software platforms to derive proportional gains in performance of the overall signature operation.

### 6.2 8051 as $\mathcal{P}_1$ and Xilinx Virtex-II FPGA as $\mathcal{P}_2$

It was noted that the performance of 8051 for point multiplication was quite poor as compared to the FPGA, (i.e., $\Theta \gg 1$). Thus only the multiprecision arithmetic was executed on 8051 and the point multiplication operation was executed in the FPGA. The results for this case are tabulated below.

| Algorithm | Time (ms) |
|---|---|
| Modular multiplication on $\mathcal{P}_1$ | 23 |
| Modular inversion on $\mathcal{P}_1$ | 120 |
| Point multiplication on $\mathcal{P}_2$ | 111.9 |

**Table 2.** Implementation Statistics for ECDSA on 8051 and Xilinx Virtex-II

With the above results, the total time for generation of a signature was 189 ms.

We note that the parallel paths consume unequal amounts of time in carrying out their computations. In particular, the software component requires 54 ms longer than the hardware. However, the complexity of point multiplication is cubic while that of multiprecision arithmetic is quadratic in the size of the field. Thus, for curves over larger fields, the point multiplication step would consume more time and the implementation would be more optimal.

The idea of parallelism can be exploited without the use of hardware. Two ARM processors can be used to speedup the signature operations

by a factor of 2. This direction of advancement seems imminent with the advent of multi-core processors.

## 7 Conclusion

In this paper, we have shown how to use two processing units of unequal speeds to implement the ECDSA signature generation algorithm. Our implementation requires only the storage of a single precomputed point. We discussed the design trade-offs and presented the implementation results that brought out the advantages of a parallel design. We showed how the algorithms for the point multiplication and modular inversion steps can be adapted to provide security from SPA attacks. Our proposed design is suitable for implementation over smart cards.

## References


1. P. Barrett, "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor", *Advances in Cryptology–CRYPTO '86*, LNCS **263**, 311–323.
2. I. Blake, G. Seroussi, and N. P. Smart, Eds. *Advances in Elliptic Curve Cryptography*, London Mathematical Society Lecture Note Series **317**, Cambridge University Press, 2005.
3. M. Brown, D. Hankerson, J. López and A. Menezes, "Software Implementation of the NIST Elliptic Curves Over Prime Fields", *Topics in Cryptology* - CT-RSA 2001, LNCS **2020** (2001), 250–265.
4. H. Cohen and G. Frey, Eds. *Handbook of Elliptic and Hyperelliptic Curve Cryptography: Theory and Practice*, CRC Press, 2005.
5. J. Coron, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems", *CHES* 1999, LNCS **1717**, 292–302.
6. H. Eberle, N. Gura and S. Chang-Shantz, "A Cryptographic Processor for Arbitrary Elliptic Curves over GF($2^m$)", *ASAP 2003*, 444–454.
7. T. ElGamal, "A Public key Cryptosystem and a Signature Scheme based on Discrete Logarithms", *IEEE Transactions on Information Theory*, **31**, (1985), 469–472.
8. W. Fischer, C. Giraud, E.W. Knudsen and J. Seifert, "Parallel scalar multiplication on general elliptic curves over $\mathbb{F}_p$ hedged against Non-Differential Side-Channel Attacks", *Cryptology ePrint Archive*, Report 2002/007.
9. J. Garcia and R. Garcia, "Parallel Algorithm for Multiplication on Elliptic Curves", *Cryptology ePrint Archive*, Report 2002/179.
10. D. Hankerson, J. Hernandez and A. Menezes, "Software implementation of elliptic curve cryptography over binary fields", *Proceedings of CHES 2000*, LNCS **1965** (2000), 1–24.
11. T. Hasegawa, J. Nakajima and M. Matsui, "A Small and Fast Software Implementation of Elliptic Curve Cryptosystems over GF($p$) on a 16-bit Microcomputer", *IEICE Trans. Fundamentals*, E82-A(1) January 1999, 98–106.

12. T. Izu and T. Takagi, "A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks", *Public Key Cryptography*, LNCS **2274**, 2002, 280–296.
13. D. Johnson and A. Menezes, "The Elliptic Curve Digital Signature Algorithm (ECDSA)", Technical Report CORR 99-34, Dept. of C&O, University of Waterloo, 1999.
14. M. Joye and J. Quisquater, "Hessian Elliptic Curves and Side-channel Attacks", *CHES* 2001, LNCS **2162**, 402–410.
15. M. Joye and C. Tymen, "Protections against Differential Analysis for Elliptic Curve Cryptography", *CHES* 2001, LNCS **2162**, 377–390.
16. M. Joye and S.-M. Yen, "The Montgomery Powering Ladder", *CHES* 2002, LNCS **2523**, 291–302.
17. D. Knuth, *The Art of Computer Programming*, Vol. 2, Seminumerical Algorithms, Third Edition, Addison Wesley, 1997.
18. P. Kocher, J. Joshua and J. Benjamin, "Differential Power Analysis", *Advances in Cryptology–CRYPTO 99*, LNCS **1666**, 388–399.
19. C. Lim and P. Lee, "More Flexible Exponentiation with Precomputation", *Advances in Cryptology–CRYPTO '94*, LNCS **839**, 95–107.
20. J. López and R. Dahab, "Fast Multiplication on Elliptic Curves over GF($2^n$) without Precomputation", *CHES* 1999, LNCS **1717**, 316–327.
21. A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
22. B. Möller, "Parallelizable Elliptic Curve Point Multiplication Method with Resistance against Side-Channel Attacks", *Information Security ISC 2002*, LNCS **2433**, 2002, 402-413.
23. P. Montgomery, "Speeding up the Pollard and Elliptic Curve Methods of Factorization", *Mathematics of Computation*, **48** (1987), 243–264.
24. J. Solinas, "Low-Weight Binary Representations for Pairs of Integers", *Tech. Report CORR 2001/41*, CACR Waterloo, 2001. Available at *http://www.cacr.math.uwaterloo.ca/techreports/2001/corr2001-41.ps*.
25. A.D. Woodbury, D. Bailey and C. Paar, "Elliptic Curve Cryptography on Smart Cards without Coprocessors", *CARDIS* 2000: 71–92.