INTERNATIONAL JOURNAL OF CIRCUIT THEORY AND APPLICATIONS
Int. J. Circ. Theor. Appl. (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cta.2283

SPECIAL ISSUE PAPER

Side-channel analysis of the modular inversion step in the RSA key generation algorithm

Alejandro Cabrera Aldaya[1,*,†], Raudel Cuiman Márquez[1],
Alejandro J. Cabrera Sarmiento[1] and Santiago Sánchez-Solano[2]

[1]*Instituto Superior Politécnico José A. Echeverría (CUJAE), Havana, Cuba*
[2]*Instituto de Microelectrónica de Sevilla, IMSE-CNM (CSIC/Universidad de Sevilla), Seville, Spain*

SUMMARY

This paper studies the security of the RSA key generation algorithm with regard to side-channel analysis and presents a novel approach that targets the simple power analysis (SPA) vulnerabilities that may exist in an implementation of the binary extended Euclidean algorithm (BEEA). The SPA vulnerabilities described, together with the properties of the values processed by the BEEA in the context of RSA key generation, represent a serious threat for an implementation of this algorithm. It is shown that an adversary can disclose the private key employing only one power trace with a success rate of 100 % – an improvement on the 25% success rate achieved by the best side-channel analysis carried out on this algorithm. Two very different BEEA implementations are analyzed, showing how the algorithm's SPA leakages could be exploited. Also, two countermeasures are discussed that could be used to reduce those SPA leakages and prevent the recovery of the RSA private key. Copyright © 2016 John Wiley & Sons, Ltd.

Received 6 May 2016; Revised 1 September 2016; Accepted 8 October 2016

KEY WORDS: side-channel analysis; SPA; binary Euclidean algorithm; RSA key generation

1. INTRODUCTION

Side-channel analysis of cryptographic algorithms is a powerful technique that can seriously compromise the security of these algorithms. Since the scientific community's early experiments with power consumption based side-channel analysis in 1999, this approach has proved to be very effective against several cryptographic algorithm implementations [1].

The observation/processing of an electronic device's power consumption when processing secret data can leak information about that data. One of the techniques that can be employed to extract secret information is simple power analysis (SPA), which exploits the existence of visually distinguishable power consumption patterns that may reveal the sequence of operations executed by an algorithm. If this sequence of operations depends on a secret value, then the implementation is considered vulnerable to SPA.

*Correspondence to: Alejandro Cabrera Aldaya, Instituto Superior Politécnico José A. Echeverría (CUJAE), Havana, Cuba.
†E-mail: aldaya@gmail.com

One of the algorithms that have been targeted using this kind of side-channel analysis is the RSA key generation procedure, an algorithm commonly used in electronic devices (i.e., hardware security modules) [2] dedicated to the generation and storage of cryptographic keys.

Side-channel leakages that may exist during the RSA key generation procedure have been analyzed in several published works, most of which have focused their attention on the algorithms employed to generate the secret prime numbers needed to construct an RSA key. In [3], the Joye and Pallier method proposed in [4] was analyzed to recover some bits of the secret primes, and the Coppersmith method was used to calculate the remaining ones, with an overall success rate of 0.1% [5]. In [6] and [7], the authors analyzed the leakages that could exist in different methods for checking the divisibility of a prime candidate by a prime sieve. Independent of the specific random prime number generation algorithms analyzed in these works and their intrinsic implementation details, the best success rate achieved was 25% [7].

This work presents a new side-channel analysis of the RSA key generation procedure. Instead of targeting a prime generation algorithm as in previous works, the proposed side-channel analysis, based on SPA leakages, focuses on the modular inversion operation required to generate an RSA private key. In this context, the RSA key generation procedure imposes the requirement that only one power consumption trace can be used to extract the private key.

There are different approaches for computing modular inverses. Among the most popular methods are those based on the extended Euclidean algorithm [8]. In order to avoid the divisions involved in this algorithm, a binary variant of it, called binary extended Euclidean algorithm (BEEA), is often preferred because it replaces multi-precision divisions with right shift operations [9]. This change results in software implementations with very good performance [10], and it is also very suitable for hardware realizations, as the required shift operations can be implemented with minimal resources [11].

The execution flow of the BEEA is highly dependent on its inputs. This property has been studied in order to identify the side-channel leakages that could compromise a secret variable processed by this algorithm. In [12], the authors proved that if an adversary knows the specific execution flow that this algorithm follows, then the algorithm's secret inputs can be disclosed. In that work, the execution flow was extracted using simple-branch prediction analysis, which is a kind of side-channel analysis intrinsic to microprocessor architectures [13].

However, the existence of power-based side-channel leakages in the BEEA has not been studied in depth by the scientific community. It was previously considered [12,14] that the BEEA is vulnerable to side-channel analysis only if all the information about its execution flow is known, but in [15], a new model was introduced that is able to establish how much information can be recovered when only some information about the execution flow is known. The same study also showed that a BEEA implementation may be vulnerable to SPA and explained how the SPA leakages of a BEEA implementation can be exploited in the context of digital signature algorithm (DSA)-like signature generation [15, 16].

This paper has the following contributions with respect to [15]: in [15], the cryptographic protocol under study was ECDSA, while here the RSA key generation procedure is analyzed. This difference is important because the BEEA's inputs in elliptic curve digital signature algorithm (ECDSA) and in RSA have different properties. These properties make that in ECDSA the number of bits that can be recovered is low in comparison with the amount that can be recovered in RSA key generation, as is formalized in Section 3. This difference implies that some analysis (not needed for ECDSA) should be performed during the processing of the power consumption traces to gather the required information as is analyzed in Sections 4 and 5. Also, in this paper, two BEEA implementations are studied allowing a more complete analysis about how the power consumption traces should be analyzed for extracting the SPA leakages that a BEEA implementation may have. On the other hand, a previously known countermeasure is proposed to be applied in a different setting with respect to its original employment without sacrificing the performance of the RSA operations.

The paper is organized as follows: Section 2 introduces the RSA key generation algorithm and describes the BEEA. Section 3 presents a side-channel analysis of the BEEA, explaining how the RSA private key can be recovered employing only one power trace. Sections 4 and 5 analyze two

BEEA implementations, illustrating how the SPA vulnerabilities are present in two very different implementations of the algorithm and showing how the implementations' power consumption traces can be processed to extract the required information. Section 6 discusses two countermeasures that can be used to eliminate SPA leakages, and finally, in Section 7, some conclusions are presented.

2. RSA KEY GENERATION ALGORITHM

The RSA cryptosystem employs two different keys for encryption and decryption. These are known as the public key and the private key, respectively. The public key is considered accessible to everyone, while the private key must remain secret. The RSA key generation algorithm defines how an RSA public key and the corresponding private key are created. The steps involved in generating an RSA key pair are shown in Algorithm 1.

```
Input: Desired key length: $nlen$
Output: public key $(N, e)$, private key $(p, q, d)$
1: Generate two odd prime numbers $p$ and $q$
2: Compute the public modulus $N = pq$
3: Compute $\phi(N) = (p-1)(q-1)$
4: Select a public exponent $e$
5: Compute the private exponent $d$ as:
    $d = e^{-1} \bmod \phi(N)$
```

Algorithm 1
RSA key generation algorithm

The desired key length (*nlen*) is the only input that this algorithm requires. This parameter defines the number of bits of the public modulus N, taking into account that an adversary must not be able to factorize it during the lifetime of the generated key. Typical values of *nlen* are 1024, 2048, and 4096 [16].

The first step in the RSA key generation algorithm is to generate two random, odd prime numbers, p and q. These randomly generated primes are used to compute the public modulus N (see step 2 of Algorithm 1). The generation of these numbers is subject to some constraints to avoid certain kinds of attacks [17]. However, the results presented in this work only require considering that p and q should have about *nlen*/2 bits, as is the case in practice to avoid some factorization attacks [17].

The parameter $\phi(N)$ (calculated at step 3 of Algorithm 1) corresponds to Euler's totient function and is only used during the key generation process, but its confidentiality must be guaranteed because if it is known, the private key could be disclosed using step 5. However, there are three facts about $\phi(N)$ that are very useful in this work and that can be known in advance:

1. The bit-length of $\phi(N)$ is equal to the bit-length of N (*nlen*).
2. Its two least significant bits are zero.
3. N and $\phi(N)$ share roughly half of their bits (the most significant ones).

The first two facts can easily be deduced considering that p and q are two prime numbers of several hundreds of bits (i.e., *nlen*/2). This means that the product of $(p-1)$ and $(q-1)$ will produce a number divisible by four and with the same bit-length as $N = pq$ (i.e., *nlen*).

According to the definition of $\phi(N)$, it can be concluded that N is a good approximation of $\phi(N)$ because p and q in (1) have about the half of the number of bits of N and $\phi(N)$.

$$\phi(N) = N - (p + q - 1) \tag{1}$$

Considering that p and q have the same bit-length, it can be concluded that the number of bits that N and $\phi(N)$ have in common is about $nlen/2$, where the exact number depends on a carry propagation. Therefore, these numbers will share their $nlen/2 - 10$ most significant bits with high probability.

The public exponent e should be selected from the interval $(1, \phi(N))$ and must be relatively prime to $\phi(N)$. For performance reasons, e is often fixed to a prime number with a low Hamming weight. For example, $e = 2^{16} + 1$ is a typical value found in many implementations that offers a good trade-off between performance and security [17, 18]. For this reason, this value of e is used to present the results of this work, although the same results can also be generalized to other values of e.

The last step of the RSA key generation algorithm aims in obtaining the private exponent d, a parameter equal to the modular inverse of e modulo $\phi(N)$. This step is the target of the proposed attack, particularly when the modular inverse is computed using the binary extended Euclidean algorithm or another algorithm based on it, as is common in practice due to its good performance. This algorithm is described in the following section, with particular attention to some of its more interesting properties.

2.1. *Binary extended Euclidean algorithm*

The binary greatest common divisor algorithm proposed by Stein in 1967 is commonly known as the binary Euclidean algorithm [9]. This algorithm can be extended (to what is known as BEEA) to obtain integers x and y such that $ax + by = gcd(a, b)$, while at the same time computing the greatest common divisor of a and b (i.e., $gcd(a, b)$). When $gcd(a, b) = 1$, the value obtained for x corresponds to the multiplicative inverse of a modulo b.

The BEEA is very useful for generating RSA keys because e and $\phi(N)$ must be relative primes. This means that the greatest common divisor of e and $\phi(N)$ must be calculated to ensure the fulfillment of this requirement; and, additionally, the multiplicative inverse of e modulo $\phi(N)$ also has to be computed. One of the BEEA's properties is that it is able to compute these quantities at the same time, offering very good performance [8,11].

In this paper, the classic variant of the BEEA was employed as it allows results to be generalized to other less common variants such as those described in [8]. Algorithm 2 shows the procedure to compute the private exponent d and the greatest common divisor of e and $\phi(N)$ employing the BEEA. In this algorithm, some steps of the original description of the BEEA (cf. [8]) used to ensure that at least one of the variables u or v is odd before step 7 were excluded. This was possible because $\phi(N)$ is an even integer and, because $gcd(e, \phi(N)) = 1$ must hold, e can never be even; it is therefore always selected to be an odd number (e.g., $e = 2^{16} + 1$), ensuring that at the first execution of step 7 of Algorithm 2, u is always odd.

A specific notation was used in this paper to describe certain properties of the BEEA. The term *u-loop* refers to the loop responsible for dividing variable u by two (steps 8 to 15 of Algorithm 2). Likewise, the term *v-loop* refers to the loop that divides v. The term *x-loop* is used to refer to either of the aforementioned two loops (where the 'x' can be 'u' or 'v'). Another important component of Algorithm 2, here denominated the *sub-step*, is associated with the subtraction operations executed between steps 24 and 31 of Algorithm 2.

Using this notation, it is possible to state certain facts about the BEEA.

Fact 1: During the first iteration, the only *x-loop* that is executed is the *v-loop* because at the start of the algorithm, u is initialized to e, which is an odd number, while v, initialized as $\phi(N)$, is always an even number, as highlighted in the previous section.

Fact 2: At any iteration, one and only one *x-loop* is executed because these loops ensure that the variables u and v are odd numbers before the execution of the *sub-step*. This means that when the *sub-step* is executed, one of its subtractions (at step 25 or at step 29) ensures that only one of variables u or v will be an even number.

```
Inputs: Integer numbers $e$ and $\phi(N)$
Outputs: $gcd(e, \phi(N))$, $d=e^{-1} \bmod \phi(N)$
1:  u = $e$
2:  v = $\phi(N)$
3:  A = 1
4:  B = 0
5:  C = 0
6:  D = 1
7:  while u != 0
8:      while even(u)
9:          u = u / 2
10:         if even(A) and even(B) then
11:             A = A / 2
12:             B = B / 2                        u-loop
13:         else
14:             A = (A + $\phi(N)$) / 2
15:             B = (B – $e$) / 2
16:     while even(v)
17:         v = v / 2
18:         if even(C) and even(D) then
19:             C = C / 2
20:             D = D / 2                        v-loop
21:         else
22:             C = (C + $\phi(N)$) / 2
23:             D = (D – $e$) / 2
24:     if u ≥ v then
25:         u = u – v
26:         A = A – C        SUBS[i]=”u”
27:         B = B – D
28:     else                                     sub-step
29:         v = v – u
30:         C = C – A        SUBS[i]=”v”
31:         D = D – B
32: return (v, $d = C \bmod \phi(N)$)
```

Algorithm 2
Binary extended Euclidean algorithm

For the sake of notation uniformity, the variables used to represent the BEEA leakages in [12] and [15] were reutilized in this study. Equivalently to their definition in [12], these variables can be defined using the *x-loop* and *sub-step* notation as follows:

$$SHIFTS[i] = \textit{the number of times that an } x - \textit{loop is executed at iteration } i.$$

$$SUBS[i] = \begin{cases} 'u' & : \quad \text{if } u \geq v \text{ at step 24} \\ 'v' & : \quad \text{if } u < v \text{ at step 24} \end{cases}$$

3. SIDE-CHANNEL ANALYSIS OF THE BEEA

In [12], a side-channel analysis of the BEEA was presented. It concluded that the algorithm's inputs can be fully recovered when the adversary is able to collect the *SHIFTS*[*i*] and *SUBS*[*i*] for all iterations. Recently, in [15], the authors proved that, knowing only the *SHIFTS*[*i*], some *SUBS*[*i*] can be recovered, and a certain number of bits of one of the BEEA inputs can be expressed as a function of the other one. With regard to power-based side-channel analysis, the latter approach is very interesting because, as commented in [15], it seems very difficult to extract the *SUBS*[*i*] directly from a power trace but much easier to extract the *SHIFTS*[*i*] if the *sub-step* can be distinguished.

Each iteration of the BEEA can be segmented into a series of *x-loop* iterations followed by a *sub-step*. This means that the *SHIFTS*[*i*] are related to the time elapsed between two consecutive *sub-steps*. Consequently, if the *sub-step* procedure can be distinguished in a power trace, then an adversary should be able to extract the *SHIFTS*[*i*]. Figure 1 shows an example of an execution flow of the BEEA where a capital 'L' is used to represent an *x-loop* iteration and a capital 'S' is used to label the *sub-step*. From this execution flow, it is possible to extract some *SHIFTS*[*i*] counting the number of 'L' between each consecutive pair of 'S'. Therefore, the first six *SHIFTS*[*i*] are as follows: 5, 2, 1, 2, 1, and 5.

If the power consumption of a BEEA implementation can be segmented like the sequence of Figure 1, then it is possible to extract the *SHIFTS*[*i*] employing SPA techniques. Sections 4 and 5 describe two different BEEA implementations and analyze their power traces, showing how the *SHIFTS*[*i*] can be extracted from a power trace for each hardware platform. In this paper, the power consumption is the source of the leakage, but the analysis of the side-channel vulnerabilities of the BEEA when it is used during an RSA key generation can be applied to any scenario where the *SHIFTS*[*i*] can be extracted. The rest of this section presents the rationale for recovering all bits of an RSA private key during its generation, considering that an adversary is able to obtain the required *SHIFTS*[*i*] using similar methods to those described in Sections 4 and 5.

According to the findings in [15], the higher the number of iterations for which the corresponding *SHIFTS*[*i*] and *SUBS*[*i*] are known, the higher the number of bits that can be recovered. More formally, when for the first T iterations the *SHIFTS*[*i*] are known and the first $T-2$ *SUBS*[*i*] are known too, then it is possible to express the $\sum_{i=1}^{T} SHIFTS[i] + 1$ least significant bits of one of the BEEA inputs (i.e., the secret input) as a function of the other input (i.e., the known input).

This model for analyzing the side-channel leakages of the BEEA can be applied in the context of RSA key generation because the value to be inverted (e) is known by the adversary. This means that some bits of $\phi(N)$ can be expressed as a function of the public exponent e, the recovered *SHIFTS*[*i*], and the corresponding *SUBS*[*i*].

In an SPA context, the recovery of the *SUBS*[*i*] seems difficult because the operations executed and the data processed on the two branches of the *sub-step* are the same, and are therefore likely to be indistinguishable in a power trace. The difference between the branches is in the order of the operands (see Algorithm 2), so differentiating between them is difficult employing SPA techniques.

On the other hand, analyzing the BEEA's inputs during the generation of the private exponent, it is observed that e and $\phi(N)$ have very different bit-lengths. That is, when e is equal to $2^{16}+1$, it has only 17 bits, while $\phi(N)$ has the same bit-length as N (usually more than 1024 bits). This means that the value of u at the start of the algorithm is equal to a 17-bit value, while v (initialized to $\phi(N)$) stores a much higher value. This difference implies that the result of the comparison at step 24 of Algorithm 2 (which determines the values of *SUBS*[*i*]) can be predicted with absolute certainty during several iterations because v will have more bits than u during those iterations. This means that for those iterations, the corresponding *SUBS*[*i*] can be inferred by an adversary

$$\mathrm{L\ L\ L\ L\ L}\boxed{\mathrm{S}}\mathrm{L\ L}\boxed{\mathrm{S}}\mathrm{L}\boxed{\mathrm{S}}\mathrm{L\ L}\boxed{\mathrm{S}}\mathrm{L}\boxed{\mathrm{S}}\mathrm{L\ L\ L\ L\ L}\boxed{\mathrm{S}}\mathrm{L\ L}$$

Figure 1. Example of a binary extended Euclidean algorithm execution flow.

(i.e., $SUBS[i]=$ 'v' as $v \gg u$). This property has devastating results on the security of a vulnerable BEEA implementation because, as is formalized below, the number of $SUBS[i]$ that can be inferred is sufficient to disclose the value of $\phi(N)$.

The exact number of iterations for which it can be concluded, with absolute certainty, that $SUBS[i]=$ 'v', depends on the number of bits that v loses at every iteration. At the ith iteration, v loses $SHIFTS[i]$ bits through the right shifts in the *v-loop*. This means that the number of iterations for which $SUBS[i]=$ 'v' is closely related to the $SHIFTS[i]$. In fact, it is very likely that it is the only reason, because the other step of the BEEA that may reduce the bit-length of v occurs at the *sub-step*, but there, u is subtracted from v, so it is very unlikely that v loses a bit at this subtraction while v has a much higher number of bits than u. The formal modeling of this behavior is not covered in this paper because, as will be shown later, the number of iterations for which it is needed to know with absolute certainty that $SUBS[i]=$ 'v' indicates that the bit-length of v will be very different from the bit-length of u. So, considering only the effect of the $SHIFTS[i]$ on the number of iterations for which it can be concluded that $SUBS[i]=$ 'v', it is clear to see that v, initialized to $\phi(N)$, has to be divided by two L times (2) to store a value of the same bit-length of u.

$$L = \log_2 \phi(N) - \log_2(e) \tag{2}$$

This indicates that the minimum number of iterations for which it can be concluded that $SUBS[i]=$ 'v' with absolute certainty is defined by the minimum value of T that satisfies $\sum_{i=1}^{T} SHIFTS[i] + 1 \geq L$. Considering the results in [15], the L least significant bits of $\phi(N)$ can be recovered if the $SHIFTS[i]$ and $SUBS[i]$ are known for the first T iterations, and taking into account that in this case L only differs by 17 from the bit-length of $\phi(N)$, then $\phi(N)$ can be almost fully recovered using this method as is shown in Figure 2. In this figure, $\phi(N)$ and N are represented as *nlen* bits numbers showing that they have in common half of their most significant bits (shaded area).

The method described earlier makes it possible to recover all bits of $\phi(N)$ except the $\log_2(e)$ most significant ones. When e is equal to $2^{16}+1$, as is common, only the 17 most significant bits of $\phi(N)$ still need to be recovered. However, as mentioned in Section 2, N and $\phi(N)$ share approximately half of their most significant bits, so the remaining 17 most significant bits (MSBs) of $\phi(N)$ are already known from the knowledge of the public parameter N. Once $\phi(N)$ is completely disclosed, the recovery of the private exponent (d) is immediate because it is known that the latter is the multiplicative inverse of the public exponent e modulo $\phi(N)$.

The number of iterations, T, for which it is needed to know the $SHIFTS[i]$ depends only on the bit-length of $\phi(N)$. Taking into account that, with high probability, N and $\phi(N)$ have their $nlen/2-10$ most significant bits in common, only $nlen/2+10$ bits of $\phi(N)$ are unknown. This means that the minimum value of T that satisfies (3) corresponds to the number of iterations for which the $SHIFTS[i]$ should be extracted from a power trace to recover the remainder bits of $\phi(N)$.

$$\sum_{i=1}^{T} SHIFTS[i] + 1 \geq \frac{nlen}{2} + 10 \tag{3}$$

At each iteration i of the BEEA, the corresponding $SHIFTS[i]$ is at least one (Section 2). Also, considering that the probability of $SHIFTS[i]=k$ is defined by $1/2^{(k-1)}$ [8], there is a 50% chance that u is divided by at least two at each iteration. This means that, on average, the first $T=nlen/4+5$ $SHIFTS[i]$ should be sufficient to fulfill (3). This leads to the conclusion that for

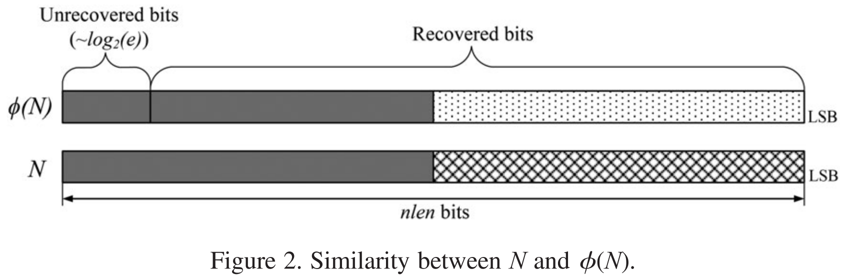

Figure 2. Similarity between N and $\phi(N)$.

RSA-2048 and RSA-4096, an average of 517 and 1029 *SHIFTS*[*i*], respectively, need to be extracted from a power trace for recovering the unknown bits of $\phi(N)$.

With regard to side-channel analysis, the bit-length difference between $\phi(N)$ and e has devastating results for the security of an RSA key generation implementation that employs the BEEA. The procedure described in this section shows that an adversary only needs to be able to extract the *SHIFTS*[*i*] to recover the private key. This renders ineffective the countermeasure proposed in [14] against an SPA attack against the BEEA designed to enforce the indistinguishability of the *sub-step* branches to avoid the recovery of the *SUBS*[*i*], because the attack presented here obtains the required *SUBS*[*i*] analytically, instead of employing SPA-based techniques.

The following section highlights the application of the attack when other values of the public exponent, e, are used, while in Sections 4 and 5, two commonly used cryptographic implementation platforms are analyzed in order to show the feasibility for extracting the *SHIFTS*[*i*] from a power trace.

3.1. Analysis for NIST standardized values of e

As shown in the previous section, the value of e (i.e., $2^{16}+1$) commonly used to speed up the RSA encryption and RSA signature verification procedures represents a serious threat when confronted by an SPA-capable adversary. Consequently, in order to reduce the number of bits of $\phi(N)$ leaked from an SPA-vulnerable BEEA implementation, e should be selected of about the same bit-length of N. However, some standards, such as NIST FIPS-186-4, restrict the selection of e to the range $2^{16} < e < 2^{256}$ [16]. This means that the maximum acceptable bit-length for e in this standard is 256. Therefore, using the side-channel leakage and the method explained previously, the 256 most significant bits of $\phi(N)$ will remain unrecovered. However, taking into consideration that the minimum security strength currently allowed is 2048 bits [19], then $\phi(N)$ and N share at least about their 1024 most significant bits, so the full recovery of $\phi(N)$ is still possible. This shows that, even with the highest value of e compliant with the NIST standard, $\phi(N)$ can be fully recovered and the private key fully disclosed during the RSA key generation procedure if an SPA-vulnerable BEEA implementation is used.

4. HARDWARE BINARY EXTENDED EUCLIDEAN ALGORITHM TARGET

In this section, a hardware implementation of the BEEA is analyzed, and some methods are presented for identifying the meaningful patterns that allow the recovery of the *SHIFTS*[*i*]. The hardware platform targeted corresponds to a third-party field-programmable gate array (FPGA) implementation of the BEEA that follows the steps of Algorithm 2. This implementation employs three embedded RAM memories (BRAMs) for storing the BEEA variables. Each variable is stored as an array of 32-bit words. This BRAM arrangement reduces latency by allowing the parallel execution of some steps of the BEEA. For example, it is known that the BEEA variables are arranged in these BRAMs in such a manner that the three subtractions at the *sub-step* stage can be executed in parallel, while in the *v-loop* implementation, the steps of the branches resulting from step 18 of Algorithm 2 are executed concurrently just after the division by two operation of the variable v. Also, at steps 22 and 23 of Algorithm 2, the divisions by two are interleaved with the addition and subtraction operations. This property is important because it could result in a different power consumption pattern with respect to the division by two and addition/subtraction ones.

4.1. Simple power analysis to a FPGA binary extended Euclidean algorithm implementation

The SAKURA-G board (Morita Tech Co., Ltd, Taipei, Taiwan) is one of the most commonly used platforms for evaluating the side-channel resistance of FPGA cryptographic algorithm implementations [20]. The BEEA core netlist was implemented for the Spartan 6 LX75 FPGA present on this board.

Power consumption traces were acquired by means of the SAKURA-G on-board amplifier output [20]. The traces were captured using a Tektronix DPO3034 oscilloscope (Tektronix, Inc., Beaverton, OR, USA) with the target implementation operating at 6 MHz [21]. Two channels of the scope were employed: one to capture the trigger signal and the other to obtain the traces. The sample rate was 250 MSps and the bandwidth was limited to 20 MHz, as this results in better looking traces.

A portion of a power trace captured using this measurement setup is shown in Figure 3. This trace corresponds to the target implementation's power consumption just at the beginning of the first iteration of the BEEA. This means that power consumption of the initialization phase (steps 1 to 6 of Algorithm 2) is not shown but, it would have a similar shape for different values of $\phi(N)$, therefore, it is not relevant in the context of an SPA.

The power trace in Figure 3 has to be analyzed to extract the *SHIFTS*[*i*] needed to recover the remainder bits of $\phi(N)$, as described in Section 3. As a *SHIFTS*[*i*] corresponds to the number of times an *x-loop* is executed at iteration *i*, it is important to identify each BEEA's iteration in the power trace. Identification of the *sub-step* procedure makes it possible to segment the BEEA power trace into iterations; because the *SHIFTS*[*i*] are related to the time elapsed between two consecutive *sub-steps,* they can therefore then be recovered.

Analyzing the power trace shown in Figure 3, it can be concluded that it is composed by a concatenation of a few power consumption patterns. These patterns are marked in the figure as **D1**, **D2**, **AD,** and **SS**. The correspondence of these patterns with the multi-precision operations (micro-operations) present in the BEEA could be directly established if it were possible to measure the power consumption of the BEEA implementation while it processes a known value of $\phi(N)$.

However, even if an adversary is unable to do this, there are alternative methods for establishing the correspondence. In this work, it is assumed that the adversary cannot control the BEEA inputs. This implies that the power consumption patterns must be identified using general facts about the processed values and the BEEA. This approach provides a more general description of the algorithm's SPA vulnerabilities.

For example, if the adversary is able to trigger the execution of an RSA key generation implementation several times, a new pair of prime numbers p and q will be generated at each execution, and a new, random $\phi(N)$ will consequently be processed by the BEEA core. Comparing a few of these traces, it is possible to note that the sequence **D1-AD-D1-D2** always appear at the start of the power trace. To identify which power consumption pattern corresponds to each of the BEEA micro-operations, some facts about this algorithm (presented in Section 2.1), together with known information about $\phi(N)$, will be used.

Taking into account that the one and only *x-loop* executed at the first iteration is the *v-loop* and that, as shown in Section 2, $\phi(N)$ is always divisible by four, then the *v-loop* will be executed at least twice during the first iteration of the BEEA (i.e., *SHIFTS*[1] ≥ 2). This leads to the conclusion that the observable sequence of patterns **D1-AD-D1-D2** should correspond to the steps of BEEA that are executed during the first two iterations of the first *v-loop*. The reason that makes this sequence of patterns always appears at the start of the BEEA is that the sequence of steps of Algorithm 2 that are executed during the first two iterations of the first *v-loop* only depends on the evenness of the variables involved (v, C, D, e, and $\phi(N)$), which are the same for every value of $\phi(N)$ during the first two iterations of the first *v-loop*.

More specifically, at the start of the first iteration of the *v-loop*, variables C and D are equal to zero and one, respectively, meaning that the BEEA will branch to steps 22 and 23 of Algorithm 2. So, C will subsequently be updated according to (4):

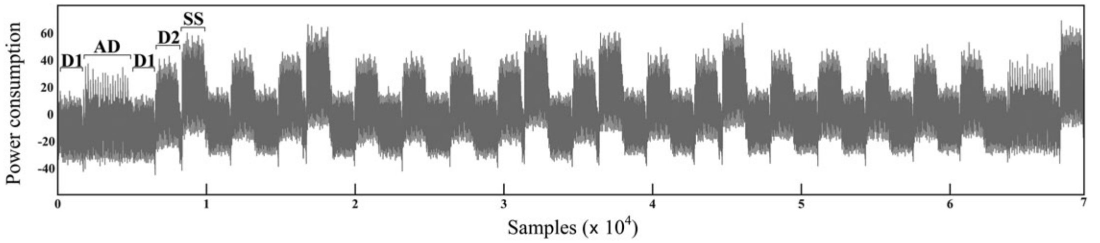

Figure 3. FPGA binary extended Euclidean algorithm implementation partial power trace.

$$C = \frac{C + \phi(N)}{2} = \frac{\phi(N)}{2} \tag{4}$$

This means, that regardless of the value of $\phi(N)$, C will be an even number, because $\phi(N)$ is always divisible by four. Performing the same analysis for variable D, which is updated using (5)

$$D = \frac{D - e}{2} = \frac{1 - \left(2^{16} + 1\right)}{2} = -2^{15} \tag{5}$$

it can be concluded that, at the start of the second iteration of the first *v-loop*, variables C and D will store even numbers, so the result of the condition at step 18 in the next iteration will also be known. This example illustrates why the sequence of steps executed by the BEEA is independent of the value of $\phi(N)$ during the first two iterations of the first execution of the *v-loop*. The fixed sequence of power consumption patterns, **D1-AD-D1-D2**, observable for different values of $\phi(N)$, should therefore correspond to the micro-operations executed during these two iterations**.**

The correspondence of the sequence of patterns **D1-AD-D1-D2** with the steps that are executed inside a *v-loop* gives the clue that the **SS** pattern should correspond to the power-consumption pattern of the *sub-step* procedure. This conclusion is also supported by the fact that the BEEA core executes some operations in parallel and, as detailed in the previous section, the inner subtractions of the *sub-step* procedure are executed concurrently. As a consequence of this parallel execution, the *sub-step* procedure can be expected to consume more power than other less concurrent operations and, as can be seen in Figure 3, the **SS** pattern is the pattern with the highest power consumption.

With this strong guess about the pattern associated with the *sub-step* procedure, it only remains to identify its instances in a power trace and measure the time that elapses between each consecutive pair of instances to obtain the required *SHIFTS*[*i*]. The **AD** pattern, however, has a different length with respect to the others, so its occurrences in the power trace must therefore be taken into account in order to avoid an erroneous extraction of a *SHIFTS*[*i*].

Analyzing the power trace shown in Figure 3, it can be stated that the **AD** pattern corresponds to steps 22 and 23 of Algorithm 2. This conclusion is supported by the easily verifiable fact that the conditional at step 18 of Algorithm 2 depends only on the evenness of the value stored in the variable D. This means that the value of D will be known for those iterations for which the extraction of *SHIFTS*[*i*] is required (i.e., those for which it can be stated that *SUBS*[*i*] = '*v*' as described in Section 3), because D is only updated at steps 23 and 31 of Algorithm 2, where in both cases the operands involved are known. Following the update of D at the first iteration (see Equation (5)), it can be seen that D should be divided by two exactly 15 times to make it an odd value. If it is assumed that the **D2** pattern corresponds to steps 19 and 20 of Algorithm 2, it must therefore appear exactly 15 times between the first two instances of the **AD** pattern and, as can be seen in Figure 3, it does. This fact is independent of the value of $\phi(N)$, so it can also be verified using different power traces acquired while the BEEA implementation processes different unknown values of $\phi(N)$.

Having identified the *sub-step* pattern (**SS**) and the conditional addition/subtraction pattern (**AD**), the *SHIFTS*[*i*] can be extracted through a simple visual inspection of the power trace, as is common in SPA-vulnerable implementations. For example, in the partial power trace shown in Figure 3, the first six *SHIFTS*[*i*] values are as follows: 2, 2, 4, 1, 2, and 6.

In addition to extracting *SHIFTS*[*i*] by visually inspecting a power trace, a signal processing tool was developed to perform this task. Analyzing the power consumption trace shown in Figure 3, it can be concluded that the patterns of interest have different means. Therefore, considering this power trace as a time series, a multiple change point detection and location algorithm can be employed to identify the instances of each power consumption pattern, especially the one belonging to the *sub-step* (**SS**). By using the binary segmentation algorithm [22] to find multiple changes in mean, the **D1**, **AD**, **D2,** and **SS** patterns can be reliably classified for this implementation. Once this analysis has been carried out, the identified mean changes are grouped together in four well-defined clusters, and the *k*-means clustering algorithm can be applied to correctly identify the location of each instance of the **D1**, **D2**, **AD,** and **SS** patterns. This analysis was performed for 1000 traces, and for all of them, the **D1**, **D2**, **AD,** and **SS** patterns were perfectly classified, recovering the required *SHIFTS*[*i*] with absolute certainty. This procedure was verified because the extracted *SHIFTS*[*i*] allow recovering each processed $\phi(N)$.

5. SOFTWARE BINARY EXTENDED EUCLIDEAN ALGORITHM TARGET

The second implementation of the BEEA that was analyzed to evaluate its SPA resistance was a software description of this algorithm for a commercial microcontroller. The targeted BEEA implementation is part of the open-source project AVR-Crypto-Lib [23].

In contrast to the hardware implementation analyzed in the previous section, the open-source nature of this library made it possible to study its source code and learn all its inner details. One difference with the description of the BEEA at Algorithm 2 noted during analysis of the source code is the way that the *x-loops* are implemented. In this library, the *x-loop* implementation follows Algorithm 3 to reduce the number of instructions employed. The main difference between these BEEA descriptions is that if an implementation strictly follows the steps of Algorithm 2, and if the conditional at step 18 is true, then the division by two of variable C is executed right after its addition to $\phi(N)$, whereas using Algorithm 3 *x-loop* variant, when the conditional at step 18 is true, two addition/subtraction operations are executed before the corresponding divisions.

```
…
16: while even(v)
17:     v = v / 2
18:     if odd(C) or odd(D) then
19:         C = C + $\phi(N)$
20:         D = D − $e$
21:     C = C / 2
22:     D = D / 2
…
```

Algorithm 3
AVR-Crypto-Lib *v-loop* implementation

Another important feature of this implementation is that the data type used to represent the BEEA variables can store numbers of arbitrary bit-length (i.e., it stores the position of the most significant bit that is equal to one). This information is used by the functions defined for this data type (i.e., BEEA operations) to avoid executing wasteful instructions. This implies that the number of clock cycles that a function spends depends on the bit-length of its inputs. This characteristic is relevant because, as described in Section 3, during the RSA key generation procedure, the variables employed in the BEEA have different bit-lengths. Thus, two instances of the power consumption pattern associated to the same mathematical operation (e.g., division by two) may not have the same time duration.

Analyzing the behavior of the BEEA variables and their bit-lengths across the execution of this algorithm, it can be concluded that the bit-length of the variable *v*, initialized to $\phi(N)$, will decrease its value at each iteration of the algorithm. Since the execution of the first iteration, variable C will be of the same bit-length of $\phi(N)$, and it will keep approximately the same bit-length throughout the execution of the algorithm. Variable D will have the bit-length of *e* from the first iteration of the BEEA, and its bit-length will never exceed this value during the execution of the algorithm. Variables A and B, initialized to one and zero, respectively, will retain these values until the *u-loop* is executed, and, as described in Section 3, this never happens for the iterations for which it is needed to extract the *SHIFTS*[*i*].

These facts are summarized in Table I and will be helpful when analyzing the power consumption trace generated by this implementation in a microcontroller as described in the following section. For each BEEA variable, the table shows the initial bit-length (just after the first *x-loop* iteration) and the

Table I. Summary of the binary extended Euclidean algorithm variables bit-length dynamics.

Variable	Initial bit-length	Trend
u	$\log_2 e + 1$	Constant
A	1	Constant
B	1	Constant
v	$\log_2 \phi(N) + 1$	Decrease
C	$\log_2 \phi(N) + 1$	Roughly the same
D	$\log_2 e + 1$	Roughly the same

variable's bit-length trend during the execution of the BEEA iterations for which it is required to extract the *SHIFTS*[i].

5.1. Simple power analysis to the AVR-Crypto-Lib binary extended Euclidean algorithm implementation

The ChipWhisperer-Lite (NewAE Technology Inc., Halifax, Canada) side-channel evaluation platform was employed to study the BEEA vulnerabilities of its implementation in the AVR-Crypto-Lib library [24]. This platform has a measurement circuit that allows the power consumption signal to be captured and transferred to a PC workstation. Tightly coupled to this measurement circuit, it also has a target board that allows rapid evaluation of side-channel resistance of cryptographic algorithms.

The target board has an Atmel XMEGA microcontroller (Atmel Corporation, San Jose, CA, USA), with an embedded 8-bit AVR microprocessor as the main processing unit [25]. The BEEA implementation of the AVR-Crypto-Lib library was compiled for this platform to capture the power consumption trace generated by this implementation on a microcontroller.

The target board was clocked at 7.37 MHz, and the measurement circuit of the ChipWhisperer-Lite was configured to sample the power consumption of the microcontroller at 29.5 MHz. This made it possible to capture four samples per microprocessor clock cycle, but as the ChipWhisperer-Lite employs synchronous sampling, this configuration allows acquiring information-relevant power traces [26].

A partial power consumption trace captured using the ChipWhisperer-Lite platform is shown in Figure 4. This trace was captured while the BEEA implementation computed the modular inverse of e modulo $\phi(N)$. Similarly to the hardware platform analyzed in Section 4, the trace shown in Figure 4 corresponds to the power consumption just at the start of the first iteration of the BEEA. The value of $\phi(N)$ used to capture this trace had 128 bits because if a typical $\phi(N)$ (e.g., higher than 2048 bits) had been used, the number of relevant power consumption patterns shown in the figure would be lower. However, in the context of SPA, this is not an issue because the structure (statistical properties) of the relevant power consumption patterns is independent of the number of bits of the operands.

Two patterns (marked as **R** and **S**) can be easily distinguished by their variance in this figure. Taking into account that this implementation of the BEEA is executed sequentially, only two power patterns are expected, corresponding to the division by two operation and the subtraction or addition operation, respectively. The comparison operations (e.g., steps 7 and 24 of Algorithm 2) can be excluded from this analysis because, as the data type used to represent the BEEA variables stores the position of the most significant bit that is equal to one, their latency is very likely to be very small in comparison with the rest.

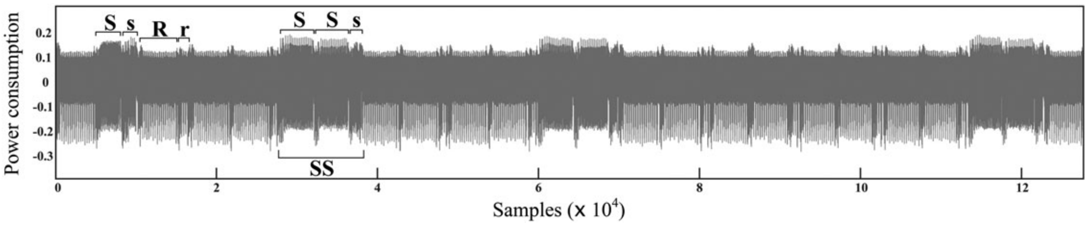

Figure 4. XMEGA binary extended Euclidean algorithm power trace.

Using this information, once the identification of which of the patterns highlighted in Figure 4 corresponds to the subtraction operation, it is a straightforward task to identify the power consumption pattern of the *sub-step* procedure because it comprises three consecutive subtraction operations.

Observing the power trace at Figure 4 and taking into account that two *sub-step* patterns cannot appear concatenated in a power trace (see Fact 2 about the BEEA in Section 2.1), it can be concluded that the **R** pattern corresponds to a division by two operation, and the **S** pattern corresponds to an addition or subtraction operation. This in turn leads to the conclusion that the power consumption pattern of the *sub-step* procedure is the one marked as **SS** in Figure 4.

It can be seen that the **SS** pattern is formed by the concatenation of two **S** patterns, and a third one (marked with a lower case **s**) that has the same variance of **S** but is shorter in length than the other two. The reason for this difference in length is that the third subtraction operation of the *sub-step* is the one that updates variable D, using $D = D - B$, and as summarized in Table I, both arguments of this subtraction have less than $\log_2 e$ bits (i.e., 17 bits for $e = 2^{16} + 1$), so the latency of this operation while processing this data can be expected to be shorter than that of the other two subtractions processing higher bit-length numbers. This difference in length also appears in the other steps where variable D is updated. In the *v-loop* description shown in Algorithm 3, at step 20, it is expected to see a subtraction pattern like the third pattern in the *sub-step* (see the first occurrence of the **s** pattern), and, at step 22, it is expected to see a division by two power consumption pattern (marked with an **r**) but shorter than, for example, the preceding one, which corresponds to the processing of a value of $\log_2 \phi(N)$ bits.

Using this information, all the *sub-step* instances can be located in a power trace, and the *SHIFTS*[*i*] can be extracted. For example, the *SHIFTS*[*i*] corresponding to the three BEEA iterations shown in Figure 4 are 2, 2, and 4.

One important phenomenon that should be taken into account is that the value of the variable v will decrease its bit-length at each iteration of the BEEA, as can be seen in Table I. Therefore, the power consumption pattern of the operations that process this variable will be shorter. This fact must be considered during the extraction of the *SHIFTS*[*i*].

Analyzing the power trace in Figure 4, it can be appreciated that the power consumption patterns of **R** and **S** differ in variance. This means that a similar time series processing to the one described in Section 4.1 can be applied, but in this case, focusing on changes in variance instead of mean. The required *SHIFTS*[*i*] are then extracted by measuring the time that elapses between two consecutive *sub-step* power consumption patterns. This step must employ the length of one *x-loop* iteration. Depending on the result of the condition at step 18 in Algorithm 3, two *x-loop* iterations can have different latencies. These latencies can be measured at the start of the power trace, because (as described in Section 4.1) both *x-loop* branches are performed during the first two iterations of the first *x-loop* execution. After that, it only needs to be remembered that the latency of the *sub-step* and each *x-loop* variant's power consumption patterns will decrease in value by the same amount because of the bit-length losses of variable v. The amount of bits that variable v loses at each iteration is known (i.e., *SHIFTS*[*i*] bits), so it is possible to know the number of power consumption samples in which two consecutive *sub-step* patterns will differ.

It is important to mention that, in addition to the method for identifying the *sub-step* pattern explained in this section, the methods described in Section 4 can also be employed, especially those that exploit general facts about BEEA behavior during the generation of an RSA private key. For example, the existence of a fixed sequence of patterns in the first two iterations of the first *v-loop* for different values of $\phi(N)$ makes it possible to distinguish all the inner operations of the *v-loop,* as explained in Section 4. This shows that, independently of implementation details, there exist several general facts about the BEEA and the values processed during the generation of an RSA key that can be used to identify the patterns of interest for extracting the required *SHIFTS*[*i*].

6. COUNTERMEASURES

This section looks at some of the countermeasures that could be used to avoid the SPA leakages described earlier in this paper. The natural countermeasure for avoiding the high number of bits that can be recovered using the proposed SPA is to use a value of e of the same bit-length of $\phi(N)$, but this change would degrade the performance of the RSA encryption and signature verification procedures, so a more efficient one is preferred.

One commonly used countermeasure to protect the value being inverted using the BEEA is the following masking procedure [27]:

1. Select r at random such that $gcd(r, \phi(N)) = 1$
2. Compute $a = e \cdot r \ mod \ \phi(N)$
3. Compute $b = a^{-1} \ mod \ \phi(N)$
4. Compute $d = r \cdot b \ mod \ \phi(N)$

In the case of RSA key generation, the secret value is the modulus ($\phi(N)$) and not the value to be inverted (e). However, this countermeasure can be used to make the value to be inverted about the same bit-length of the modulus to avoid a massive recovery of its least significant bits. In [15], it is shown that if both BEEA's inputs are of the same bit-length, the probability of recovering 20 bits is approximately 0.0048, making impractical the recovery of the amount of bits needed to compromise the value of $\phi(N)$.

The value of r must be selected in such a manner that $gcd(r, \phi(N)) = 1$ holds. This can be fulfilled regardless of the value of $\phi(N)$, by selecting r as a prime number of *nlen* bits. This method is valid because $\phi(N)$ is always divisible by four, so it means that it does not have a divisor of its same bit-length. It ensures that $gcd(r, \phi(N)) = 1$ for every value of $\phi(N)$ and prime number r of *nlen* bits.

The prime number r could be a fixed value selected during the development stage and hardcoded in the system. The application of this countermeasure does not cause a significant performance loss in the RSA key generation procedure, but it does make it possible to use a value of e with a low Hamming weight to speed up other RSA computations. It is also recommended that the fixed value r should remain secret to avoid hypothetical side-channel attacks that might exploit knowledge of one of the BEEA inputs [15]. With this countermeasure, the modular inversion step is protected against SPA, but for making the whole RSA key generation procedure secure against side-channel attacks, all its steps should be analyzed and protected if needed.

7. CONCLUSIONS

The SPA vulnerabilities of the BEEA described in this paper represent a serious threat for an implementation of the RSA key generation procedure. This is due to the bit-length difference between e and $\phi(N)$, which allows the recovery of $\log_2\phi(N) - \log_2(e)$ bits of $\phi(N)$ with absolute certainty employing only one power consumption trace.

The fact that $\phi(N)$ and N share half of their most significant bits is very useful for this research because, as the SPA vulnerability allows an adversary to recover the least significant bits of $\phi(N)$, knowledge of the most significant bits considerably reduces the number of iterations of the BEEA for which the SPA information must be gathered to recover all bits of $\phi(N)$. This allows concluding that, with high probability about $nlen/4 + 5$ BEEA iterations, on average, need to be processed to extract the required $SHIFTS[i]$ from a power trace. This means that for recovering a 4096-bit private key, about 1029 iterations should be processed to obtain the private key.

The existence of the described SPA vulnerabilities was demonstrated in two different BEEA implementations. One of the targets was an FPGA parallel implementation of this algorithm, while the other was a software implementation of the BEEA in an 8-bit microcontroller. In both cases, it was shown that it is possible to identify each BEEA's iteration and therefore recover all the $SHIFTS[i]$ required to succeed by means of visual inspection and a statistical processing of the corresponding power consumption trace. During the analysis of these different implementations, it was shown how general facts about the BEEA and the values that it processes during the generation of an RSA private key facilitate identification of the power consumption patterns of interest, regardless of inner implementation details.

The application of the multiplicative masking countermeasure of the value to be inverted by a BEEA implementation can be employed to eliminate the bit-length difference between this value and the modulus. This countermeasure prevents the recovery of the unknown bits of $\phi(N)$ with minimal impact on RSA key generation performance.

ACKNOWLEDGEMENTS

This work has been partially funded by the projects TEC2014-57971-R and RTC-2014-2932-8 from the Spanish Government (with support from FEDER).

REFERENCES

1. Kocher P, Jaffe J, Jun B. Differential power analysis. In *Advances in Cryptology (CRYPTO)*. Springer: Berlin Heidelberg, 1999.
2. Attridge J. *An Overview of Hardware Security Modules*. SANS Institute: Bethesda, MD, USA, 2002.
3. Clavier C, Coron JS. On the implementation of a fast prime generation algorithm. In *Cryptographic Hardware and Embedded Systems*. 2007.
4. Joye M, Paillier P, Vaudenay S. Efficient generation of prime numbers. In *Cryptographic Hardware and Embedded Systems*, Paar C, Koç ÇK (eds). Springer: Berlin Heidelberg, 2000.
5. Coppersmith D. Finding a small root of a bivariate integer equation; factoring with high bits know. In *EUROCRYPT*, Maurer UM (ed). Springer: Heidelberg, 1996.
6. Finke T, Gebhardt M, Schindler W. A new side-channel attack on RSA prime generation. In *Cryptographic Hardware and Embedded Systems*, Clavier C, Gaj K (eds). Springer: Berlin Heidelberg, 2009.
7. Bauer A, Jaulmes E, Lomné V, Prouff E, Roche T. Side-channel attack against RSA key generation algorithms. In *Cryptographic Hardware and Embedded Systems*, Batina L, Robshaw M (eds). Springer: Berlin Heidelberg, 2014.
8. Knuth DE. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 1997.
9. Stein J. Computational problems associated with Racah algebra. *Journal of Computational Physics* 1967; **1**(3):397–405.
10. OpenSSL Development Community. OpenSSL: the open source toolkit for SSL/TLS. 2014. [Online]: http://www.openssl.org.
11. Brent RP. Systolic VLSI arrays for linear time GCD computation. *VLSI'83* 1983:145–154.
12. Acıiçmez O, Gueron S, Seifert J-P. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *Cryptography and Coding*. Springer: Berlin Heidelberg, 2007.
13. Acıiçmez O, Koç ÇK, Seifert J-P. On the power of simple branch prediction analysis. In: *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*. ACM, 2007.
14. Aravamuthan S, Thumparthy VR. A parallelization of ECDSA resistant to simple power analysis attacks. In: *2007 2nd International Conference on Communication Systems Software and Middleware*. Institute of Electrical & Electronics Engineers (IEEE), 2007 doi:10.1109/comswa.2007.382592.
15. Cabrera Aldaya A, Cabrera Sarmiento AJ, Sánchez-Solano S. SPA vulnerabilities of the binary extended Euclidean algorithm. *Journal of Cryptographic Engineering* 2016; **accepted**. doi:10.1007/s13389-016-0135-4.
16. National Institute of Standards and Technlogy (NIST). Digital signature standard (DSS). 2013.
17. Hinek MJ. *Cryptanalysis of RSA and Its Variants*. CRC Press: Taylor & Francis Group, 2010.
18. Boneh D. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS* 1999; **46**(2):203–213.
19. Giry D. Cryptographic Key length Recommendation1. BlueKrypt. 2016. Jan 2016. [Online]: http://www.keylength.com/en/compare/.
20. Morita Tech Co. SAKURA-G development board. 2013. [Online]: http://www.morita-tech.co.jp/SAKURA/en/index.html.
21. Tektronix. Mixed signal oscilloscopes: MSO3000, DPO3000 series datasheet. 2013. [Online]: http://www.tek.com/sites/tek.com/files/media/media/resources/MSO3000-DPO3000-Mixed-Signal-Oscilloscope-Datasheet-11.pdf.
22. Killick R, Eckley IA. Change point: an R package for changepoint analysis. *Journal of Statistical Software* 2014; **58**(3):1–19.
23. Otte D, et al. AVR Crypto Lib. Aug 2015. [Online]: http://avrcryptolib.das-labor.org/trac.
24. O'Flynn C. Kickstarter for ChipWhisperer-Lite. 2015. May 2015. [Online]: ttps://www.kickstarter.com/projects/coflynn/chipwhisperer-lite-a-new-era-of-hardware-security/.
25. Atmel Corp. Atmel AVR XMEGA a 8-bit microcontroller – user guide. 2009. [Online]: http://www.atmel.com/images/doc8077.pdf.
26. O'Flynn C, Chen Z. Synchronous sampling and clock recovery of internal oscillators for side channel analysis and fault injection. *Journal of Cryptographic Engineering* 2014; **5**(1):53–69.
27. Galindo D, Großschädl J, Liu Z, Vadnala PK, Vivek S. Implementation of a leakage-resilient ElGamal key encapsulation mechanism. *Journal of Cryptographic Engineering* 2016; **6**(3):229–238.