Generating Provable Primes Efficiently on Embedded Devices

Christophe Clavier[1], Benoit Feix[1,2], Loïc Thierry[2,⋆], and Pascal Paillier[3]

[1] XLIM, University of Limoges,
`christophe.clavier@unilim.fr`

[2] INSIDE Secure
`bfeix@insidefr.com,thierry.loic@hotmail.fr`

[3] CryptoExperts
`pascal.paillier@cryptoexperts.com`

Abstract. This paper introduces new techniques to generate provable prime numbers efficiently on embedded devices such as smartcards, based on variants of Pocklington's and the Brillhart-Lehmer-Selfridge-Tuckerman-Wagstaff theorems. We introduce two new generators that, combined with cryptoprocessor-specific optimizations, open the way to efficient and tamper-resistant on-board generation of provable primes. We also report practical results from our implementations. Both our theoretical and experimental results show that constructive methods can generate provable primes essentially as efficiently as state-of-the-art generators for probable primes based on Fermat and Miller-Rabin pseudo-tests. We evaluate the output entropy of our two generators and provide techniques to ensure a high level of resistance against physical attacks. This paper intends to provide practitioners with the first practical solutions for fast and secure generation of provable primes in embedded security devices.

Keywords: Prime Numbers, Pocklington's theorem, Public Key Cryptography, Embedded Software, Modular Exponentiation, Cryptographic Accelerators, Primality Proving.

1 Introduction

Large prime numbers are a basic ingredient of keys in several standardized primitives such as RSA [21], Digital Signature Algorithm (DSA) [12] or Diffie-Hellman key exchange (DH) [10]. This paper precisely addresses the generation of provable prime numbers in embedded, crypto-enabled devices.

When it comes to RSA key generation, two approaches coexist: key pairs may be generated off-board (i.e. out of the device) in a secure environment such as a certified Hardware Security Module (HSM) running in a personalization center, and loaded into devices afterwards. Key pairs may also be generated on-board,

⋆ Part of this work was carried out when the author was doing his Master's thesis at Inside Secure.

that is, by the device itself. In this case the private key cannot be compromised as it is never transmitted to the outside world. This capability also allows the device to generate new keys later on, when deployed in the field. However it implies that the device must be able to generate large primes very efficiently and in a side-channel-secure manner.

Surprisingly enough, in spite of a quite abundant literature on primality testing and on the validation of provable primes, research works that specifically suggest generators for embedded devices are pretty inexistant. Commonly found prime number generators rely on primality (pseudo-)tests to provide a high level of confidence that the output number is prime. It is widely known that this confidence level can be increased arbitrarily by applying sufficiently many iterations of the Miller-Rabin test [12].

Technical requirements for the generation of prime numbers well-suited for RSA, DSA and ECDSA are described in industry standards such as FIPS 186-3 [12]. To ensure compliance, generating a 1024-bit DSA prime number requires as many as 40 Miller-Rabin iterations, which can be reduced to 3 when performing an additional Lucas test. However carrying out a Lucas test is more costly on an embedded device than a single modular exponentiation, and thus leads to a performance loss. This paper investigates another approach, namely the application of constructive techniques to achieve truly provable primality.

In this paper, we introduce two efficient methods for generating provable primes and present fast implementations of these methods on a popular smartcard cryptoprocessor. Our methods rely on Pocklington's theorem and an extended result due to Brillhart, Lehmer and Selfridge. We establish bounds on the entropy of the output distribution of each method and provide evidence that both of them are secure and can be used for cryptographic purposes. Performance measurements are given that demonstrate the efficiency of our algorithms and how they compare with probable prime generation. We also suggest a number of countermeasures against state-of-the-art side-channel and fault-based analysis to ensure security in an untrusted environment.

Roadmap. Section 2 recalls the usual methods for primality testing, where we distinguish between probabilistic and true tests. Generation algorithms for provable primes are discussed in Section 3, where we introduce our two efficient constructive methods. The security of these methods in terms of output entropy is discussed in Section 4. Practical results are reported in Section 5 together with performance comparisons for smartcard implementations of our probable prime and provable prime generators. Section 6 addresses threats arising from side-channel attacks and shows how to adapt our algorithms to resist these. We conclude in Section 7.

2 Prime Number Generation based on Primality Testing

In the broadest possible sense, a primality test $\top$ is a procedure that outputs a guess $\top(n) \in \{\mathsf{true}, \mathsf{false}\}$ as to whether a positive integer n is prime or composite. It can be a pseudo-primality test (also called compositeness test), in which case the guess can be a false positive with some probability, or a true primality test that never fails and provides a proof for primality when positively answered. Once one is given some primality test $\top$, it is natural to derive Algorithm 2.1 which provides a generic method for generating prime numbers.

Alg. 2.1 Generic Prime Number Generation

```
Input: a primality test $\top$, a constraining property $\mathcal{P}$
Output: a prime integer $n$
1. generate a random candidate $n$ verifying property $\mathcal{P}$
2. while $\top(n) =$ false do
3.     update $n$ while preserving property $\mathcal{P}$
4. return $n$
```

Following the naming of Brandt and Damgård [18], we refer to the list of tested candidates as the *search sequence.* In the generic prime number generator, each candidate along the search sequence is required to verify some property $\mathcal{P}$. The purpose of this requirement is to reduce the average number of calls to $\top$, which is assumed to be the most time-consuming subroutine of the algorithm, by avoiding candidates known to be composite.

Without this requirement – or equivalently, when $\mathcal{P}$ is satisfied for any n – the average number of calls to $\top$ when generating an ℓ-bit prime is close to $\ln(2^\ell)$. An obvious improvement is to let $\mathcal{P}$ be the property that n is odd and proceed to updating a candidate by adding 2 to it. In that case the average number of calls to $\top$ drops to $\ln(2^\ell)/2$. A straightforward generalization of this idea is to take for $\mathcal{P}$ the property that n is relatively prime with the t smallest primes $p_1, \ldots, p_t$. The first candidate in the search sequence thus requires the generation of an invertible element modulo $\Pi = \prod_{i=1}^{t} p_i$, which can be done either with trial divisions by each of $p_1, \ldots, p_t$, using Chinese remaindering (*e.g.* Garner [13] or Gauss algorithms), or using a technique due to [16] based on Carmichael's theorem. Several methods can be applied to update n while preserving $\gcd(n, \Pi) = 1$; Π can simply be added to n, or one can keep track of an array of indicators $\omega_i = n \bmod p_i$ for $i = 1, \ldots, t$ and modular-add 2 to all of those until none is equal to zero. Alternately, an efficient method for preserving $\gcd(n, \Pi) = 1$ for maximally large Π is found in Joye et al. [15,16]. Overall, the techniques described in [15,16] provide the most efficient approach on a cryptoprocessor as they generate an invertible element modulo Π faster than the classical trial division method. Irrespective of the chosen methods to implement the different subroutines of Algorithm 2.1, the average number of calls to $\top$ is close to

$$N(\ell, \Pi) = \ln(2^\ell) \cdot \frac{\phi(\Pi)}{\Pi}$$

where ϕ is Euler's function. The optimal choice therefore consists in taking the largest possible prime product $\Pi = p_1 \cdot \cdots \cdot p_t$. While $N(\ell, \Pi)$ obviously further decreases with larger t, the relative gain rapidly decreases as well as Π becomes larger.

2.1 Pseudo-Primality Tests

Pseudo-primality tests may erroneously view a composite number as being prime. Among these, Fermat and Miller-Rabin tests are the most commonly used in embedded applications as they are particularly fast and easy to implement. The random-base Miller-Rabin test has an error probability $\varepsilon < 1/4$. By iterating this test h times with different random bases this probability is (often quite loosely) upper bounded by $1/4^h$. Practitioners choose the number h of iterations depending on the bitsize of the tested number, the cryptosystem intended to make use of the generated prime, and the specific security requirements imposed by industry standards. Referring to FIPS 186-3, a 1024-bit prime to be used as a DSA parameter requires 40 Miller-Rabin tests (or 3 Miller-Rabin tests followed by a Lucas test). For a 2048-bit RSA key, each 1024-bit prime must pass 4 Miller-Rabin tests, and although applying the Lucas test is not required, it is highly recommended. The random-base Fermat test has approximately the same efficiency as the random-base Miller-Rabin test while its error probability is higher. However, it is more simple to implement and leads to optimally efficient pseudo-testing when using a base fixed to 2: modular multiplications by 2 can then be replaced with modular additions in the modular exponentiation $2^{n-1} \bmod n$. Fermat testing is usually performed first with $a = 2$, and only when n passes the Fermat test, does it undergo several Miller-Rabin rounds with random bases before being considered to be prime. This leads to the efficient prime number number generator referred to as Algorithm 2.2, where $\mathsf{F}_a(n)$ and $\mathsf{MR}_a(n)$ respectively denote Fermat and Miller-Rabin tests with base a.

Alg. 2.2 Efficient Generation of Probable Primes

```
Input: a bitsize $\ell$, $\Pi = 2 \cdot 3 \cdot 5 \cdot \ldots \cdot p_t$, a confidence parameter $h$
Output: an $\ell$-bit probable prime $n$
1. generate a random $\ell$-bit integer $n$ with $\gcd(n, \Pi) = 1$ and go to 3
2. update $n$ such that $\gcd(n, \Pi) = 1$
3. if $\mathsf{F}_2(n) =$ false then go to 2
4. for $i = 1$ to $h$ do
5.     pick a base $a$ at random from $[2, n-2]$
6.     if $\mathsf{MR}_a(n) =$ false then go to 2
7. return $n$
```

Neglecting the probability that the output prime is a Fermat or a strong pseudoprime, and denoting respectively by T_i, T_u, T_{F_2} and T_{MR_a} the execution times of the routines for generating the first candidate, updating the current candidate and performing Fermat and Miller-Rabin tests, the average total execution time to generate a probable l-bit prime amounts to

$$T_{\text{probable}}(\ell) = T_i(\ell) - T_u(\ell) + N(\ell, \Pi) \cdot (T_u(\ell) + T_{\mathsf{F}_2}(\ell)) + h \cdot T_{\mathsf{MR}_\mathsf{a}}(\ell) \ . \quad (1)$$

This generation method is among the most popular ones in use in the embedded security industry at the present time. Section 5 reports practical performance figures for a typical smartcard implementation of this generator.

2.2 True Primality Tests

Prime number generators make use of pseudo-primality tests because of their efficiency. However, to fully eliminate the error probability ε, one has to rely on true primality testing a.k.a. primality proving. The asymptotically fastest true primality test is the AKS method [1], which is the only known algorithm that runs in polynomial time. However, the preferred general-purpose method for testing large numbers is currently the Elliptic Curve Primality Proving test [4] which was used to ascertain the primality of the largest general number, a prime with more than $20'000$ decimal digits. Unfortunately the AKS and ECPP methods are way too complex to be of any interest for embedded implementations, where algorithms are preferably based on simple arithmetic operations such as modular exponentiations.

A possible step in this direction relates to a deterministic variant of the Miller-Rabin criterion. Following a result from Ankeny [3], Bach [5] proved under the Extended Riemann Hypothesis (ERH) that any composite number n has a strong witness[4] upper bounded by $2\ln^2 n$. Thus, verifying that n passes Miller-Rabin testing for all bases smaller than $2\ln^2 n$ would actually prove that n is prime. The drawback of this approach is the fairly large amount of bases to consider before making sure that n is prime. Proving the primality of a 512-bit number would require more than $250'000$ Miller-Rabin rounds. A secondary drawback is that the primality proof only holds under ERH.

Instead of relying on the existence of a small witness, it may be better to rely on the existence of a small set containing at least one witness. Given an upper bound x on candidates, a *reliable set* of witnesses is a set $\mathcal{W}$ such that every odd composite integer $n \leq x$ has a witness in $\mathcal{W}$. An interesting result from Alford et al. [2] unconditionally proves the existence of a reliable set containing at most $(6/5)\ln x$ integers smaller than x. This result does not rely on any conjecture and proves that n is prime with much fewer Miller-Rabin rounds (only 426 rounds for 512-bit numbers). Unfortunately the constructive method put forward by the authors for identifying such a reliable set does not seem to be computationally practical.

3 Constructive Generation of Provable Primes

As previously discussed, there does not seem to be any practical true primality test that would suit our context. Rather than testing the true primality of

[4] A *strong witness* for a composite number n is an integer a such that n does not pass the Miller-Rabin test with base a, thereby proving its compositeness.

candidates along a search sequence, we revisit Maurer's approach [18] wherein provable primes are generated in a *constructive* manner using Pocklington's criterion:

Theorem 1 (Pocklington's theorem). *Let $n > 3$ be an odd integer, and let $n = rF + 1$ where the factorization of F is known as $F = \prod_{j=1}^{s} q_j^{e_j}$. If there exists an integer a such that*

(i) $a^{n-1} \equiv 1 \pmod{n}$ *and*
(ii) $\gcd(a^{(n-1)/q_j} - 1, n) = 1$ *for each* $j = 1 \ldots s$*,*

then every prime divisor p of n is congruent to 1 modulo F. In particular, if $F > \sqrt{n} - 1$ then n is prime.

As opposed to Fermat and Miller-Rabin's theorems, Pocklington's theorem isolates sufficient conditions for true primality. Unfortunately it cannot be used to test any given integer since the factorization of $n-1$ must be partially known. Based on Pocklington's theorem, Maurer [18] suggested a constructive method for generating provable primes. The main idea there is to construct a prime n such that $n-1$ is divisible by one or more smaller primes. A recursive use of the criterion then allows to generate larger primes at each round starting from small integers whose primality proof is trivial.

Theorem 2. *Let p be an odd prime, and r an integer such that $r < p$. Let $n = 2rp + 1$.*

(i) If there exists an integer a with $2 \leq a < n$ such that $a^{n-1} \equiv 1 \pmod{n}$ and $\gcd(a^{2r} - 1, n) = 1$ then n is prime.
(ii) If n is prime, the probability that a random value a satisfies $a^{n-1} \equiv 1 \pmod{n}$ and $\gcd(a^{2r} - 1, n) = 1$ is $1 - 1/p$.

A generation algorithm can be derived from Theorem 2 (i) by iteratively producing provable primes twice larger at each iteration. Maurer proposed an iterative (and recursive) provable generation method based on this approach [19]. This iterative method requires precomputing and storing the intermediate bitsize of all provable primes from the highest to the lowest. In Maurer's algorithm, the number of iterations is variable and depends on a parameter r which is computed in order to provide the best output entropy. The main drawback of this implementation is that it is not efficient enough and therefore not suited to embedded implementations.

3.1 The Square Root Method

We now show how to generate provable primes more efficiently using Theorem 2 with fixed bitsizes for intermediate primes. We generate a provable prime by doubling at each iteration the size of the current prime p to derive the new prime $n = 2rp + 1$. While the entropy of this approach – estimated later in the paper – is not as optimized as in Maurer's algorithm, this offers a more suitable and efficient algorithm in embedded environments.

The intermediate prime sizes can be seen as equivalent to those in Maurer's algorithm when fixing $r = 0.5$. An iterative and recursive method relying on this idea – doubling each time the size of primes – was also proposed by Shawe-Taylor in [22] before Maurer's publication and is recommended by the NIST [12] to generate provable primes for public key schemes. The first algorithm we propose can therefore be seen as an adaptation of the Shawe-Taylor method, which also relies on Pocklington's theorem. As opposed to Shawe-Taylor, our algorithm is not recursive but directly generates the primes iteratively from the smallest to the largest and many additional optimizations are put forward to improve efficiency.

Initialization. Before making use of Pocklington's theorem, one starts the generation with a first prime with initial bitsize l_0. In his algorithm, Maurer suggests generating the first prime (which is 20-bit long in the best case) using Erathostene's sieve. Our approach here is different and applies the Miller-Rabin criterion to generate initial primes up to 2^{32}. Indeed, Pomerance et al. [20] and Jaeschke [14] have proven that any number lesser[5] than 2^{32} is proven prime if it successfully passes the Miller-Rabin test with the three bases 2, 7 and 61. Making use of this trick, we obtain the algorithm $\mathsf{InitGenPrime}(\ell_0)$ (given in Appendix A). We define the bitsize of the initial prime as

$$\ell_0 = \min_{k>0} \left\{ \left\lceil \frac{\ell_n - 1}{2^k} \right\rceil + 1 \text{ such that } \left\lceil \frac{\ell_n - 1}{2^{k-1}} \right\rceil + 1 > 32 \right\} .$$

As indicated previously, we make use of $\mathsf{InitGenPrime}(\ell_0)$ to generate the initial prime p for any given size ℓ_0 lesser than 32. To illustrate the different steps of our method, Table 1 gives for different bitsizes ℓ_n, the initial prime size ℓ_0, the number k of iterations of Pocklington's theorem, and the intermediate prime sizes ℓ_i at each iteration.

ℓ_n	k	ℓ_0	ℓ_1	ℓ_2	ℓ_3	ℓ_4	ℓ_5	ℓ_6	ℓ_7
512	5	17	33	65	129	257	512	-	-
768	5	25	49	97	193	385	768	-	-
1024	6	17	33	65	129	257	513	1024	-
2048	7	17	33	65	129	257	513	1025	2048

Table 1. Intermediate bitsizes (ℓ_0 and ℓ_i) and number k of iterations.

In order to reduce the number of Fermat tests throughout the generation, we apply the same idea as in the generation of probable primes: we get rid of

[5] More precisely, the exact bound is $4'759'123'141$.

candidates n which are not coprime to a product Π of the smallest primes. We thus obtain the provable prime generator presented as Algorithm 3.1.

Alg. 3.1 Efficient-Square-Root-Generation(ℓ_n)

```
Input: a bitsize $\ell_n$, $\Pi = 3 \cdot 5 \cdot \ldots \cdot p_t$
Output: an $\ell_n$-bit provable prime $n$
 1. $\ell \leftarrow \ell_n$
 2. while $\ell > 31$ do
 3.     $\ell \leftarrow \ell/2$
 4. $\ell \leftarrow \ell + 1$
 5. $n \leftarrow \mathsf{GenInitPrime}(\ell)$                    [compute the initial small prime]
 6. while $\ell < \ell_n$ do
 7.     $p \leftarrow n$
 8.     $\ell \leftarrow \min(2\ell - 1, \ell_n)$
 9.     $I \leftarrow \lfloor \frac{2^{\ell-1}}{2p} \rfloor$
10.     Select $r$ at random from $[I+1, 2I]$ such that $n \leftarrow 2rp + 1$ is coprime to $\Pi$ and
        go to 12
11.     Update $r$ in $[I+1, 2I]$ such that $n \leftarrow 2rp + 1$ is coprime to $\Pi$
12.     if $\ell < 129$ then
13.         pick an integer $a$ at random from $[2, n-2]$
14.     else
15.         $a \leftarrow 2$
16.     if $a^{n-1} \bmod n \neq 1$ then go to 11
17.     if $\gcd(a^{2r} - 1, n) \neq 1$ then go to 11
18. return $n$
```

Selection and Update of *r* and *n*. A first solution for finding a suitable r at Step 10 of Algorithm 3.1 consists in randomly selecting a first value $r \in [I+1, 2I]$, setting $n = 2rp + 1$, and then incrementing r by 1 and n by $2p$ until the modular residues $(\omega_i = n \bmod p_i)_{i=1,\ldots,t}$ are all non zero. Each ω_i is then incremented by $2p \bmod p_i$. An efficient trick consists in obtaining the values $2p \bmod p_i$ by doubling modulo p_i the residues ω_i of the previous iteration since the previous value of n corresponds to the new value of p in the current iteration. At Step 11, the same incremental update of r and n is applied for generating the next candidate coprime to Π.

A second solution consists in generating n simultaneously compliant with Pocklington's property (an even multiple of p plus one) and coprime to Π. This is done by first selecting r as $(x - (2p)^{-1} \bmod \Pi)$ where x is randomly selected from $\mathbb{Z}^\star_\Pi$ using the technique of [15] based on Carmichael's function. Then r is added to a random multiple of Π so that it lies in $[I+1, 2I]$, and the first candidate n is computed as $2rp+1$. Doing so, n is constructively coprime to Π. At Step 11, the next candidate is computed in the same vein from the updated value $x \leftarrow p_{t+1} \cdot x \bmod \Pi$.

Fixing $a = 2$ in Fermat testing. From Theorem 2 (ii), we know that the probability that a random value a rejects a prime n at Step 16 or 17 is $1/p$. Assuming that the fraction of rejected primes does not vary much from one value of a to another, choosing a constant value a has a negligible impact on the distribution of the generated primes when the bitsize ℓ is sufficiently large. For instance when generating a 128-bit prime number $n = 2rp + 1$ from a 65-bit provable prime p, less than $1/2^{64}$ of the primes would never be reached. We accept this negligible loss of entropy and use $a = 2$ for the Fermat test when $\ell > 128$. This leads to faster exponentiations for steps 16 and 17 where modular multiplications by the base can be replaced with modular additions.

Estimated Performance. Denoting respectively by T_{init}, T_I, T_u, T_{F_a} and T_g the execution times taken by the initialization, computing I, updating the candidate n, the Fermat test with base a and the gcd computation, the total average execution time of Algorithm 3.1 amounts to

$$T_{\text{provable}}(\ell_n) = T_{\text{init}}(\ell_0) + \sum_{i=1}^{k} \left(T_I(\ell_i) + N(\ell_i, \Pi) \cdot (T_u(\ell_i) + T_{\mathsf{F}_a}(\ell_i)) + T_g(\ell_i)\right) . \quad (2)$$

We report experimental results from our smartcard implementation of this prime number generator in Section 5. Note that the value $N(\ell_i, \Pi)$ equals the average number of primality tests in the generation of probable primes for ℓ_i-bit integers coprime to Π. Also, as expected, we observed in our simulations that only one gcd is computed per ℓ_i-bit prime so that its execution time is almost negligible compared to the overall execution time.

3.2 The Cube Root Method

Our second method relies on (what we refer to) as the Cube Root Theorem put forward by Brillhart, Lehmer and Selfridge in 1970. More details on this result can be found in [6].

Theorem 3 (Brillhart-Lehmer-Selfridge-Tuckerman-Wagstaff [6]). *Let $n > 3$ be an odd integer, let $n = rF + 1$ where F is completely factored and $gcd(F, r) = 1$. Suppose there exists an integer a such that*

(i) $a^{n-1} \equiv 1 \pmod n$,
(ii) $\gcd(a^{(n-1)/q} - 1, n) = 1$ *for each prime factor q of F.*

Let $r = uF + s$, $1 \le s < F$, and suppose $n < 2F^3 + 2F$, $F > 2$. If u is odd, or if u is even and $s^2 - 4u$ is not a perfect square, then n is prime.

As a corollary of Theorem 3, we derive the following result:

Theorem 4 (Cube Root Theorem). *Let p be an odd prime, $n = 2rp+1$ with r an integer such that $r < p^2 + 1$. If there exists an integer a with $2 \le a \le n$ such that* *(i)* $a^{n-1} \equiv 1 \pmod{n}$ *and* $\gcd(a^{2r}-1, n) = 1$,

(ii) $r = up + s$, $1 \leq s < p$ *for odd* u,

then n *is prime.*

Theorem 4 makes it possible to put together a prime number generator that iteratively produces proven primes three times larger at each iteration (instead of twice larger in the Square Root method). In order to speed-up the whole generation, we only consider cases where the quotient u is odd. This reduces the output entropy by one bit but has no significant impact on the security of cryptosystems such as RSA and DSA. To generate a provable prime of ℓ_n bits, our algorithm starts with the generation of an initial prime p of ℓ_0 bits, where ℓ_0 is established as follows:

```
$\ell_0 \leftarrow \ell_n$
while $(\ell_0 > 31)$    $\ell_0 \leftarrow \lfloor \ell_0/3 \rfloor + 1$
```

The generation of this ℓ_0-bit initial prime is performed as previously using the Miller-Rabin criterion and algorithm $\mathsf{InitGenPrime}(\ell_0)$ of Appendix A. The sizes ℓ_i of intermediate primes are displayed on Table 2.

ℓ_n	k	ℓ_0	ℓ_1	ℓ_2	ℓ_3	ℓ_4
512	3	20	59	176	512	-
768	3	29	86	257	768	-
1024	4	14	41	122	365	1024
2048	4	26	77	230	689	2048

Table 2. Intermediate sizes (ℓ_0 and ℓ_i) and number k of iterations.

We then obtain the Cube Root prime number generator described in Algorithm 3.2.

Initial Selection and Update of r and n. A first solution for selecting a suitable r at Step 10 of Algorithm 3.2 is similar to the one used in the Square Root algorithm 3.1. An additional step is necessary that consists in computing u and s in $r = up + s$ in order to avoid candidates for which u is even.

Our second and most efficient solution for Step 10 consists in generating n in a constructive manner so that n is simultaneously compliant with Pocklington's requirement (an even multiple of p plus one), is coprime to Π and such that the quotient $u = \lfloor r/p \rfloor$ is forced to be odd. To this end, we keep track of an invertible element $x \in \mathbb{Z}_\Pi^\star$ which will serve as the residue of n modulo the prime product Π, and set $r = x - 1/(2p) \bmod \Pi$ to ensure that $n = 2xp \bmod \Pi$ is invertible modulo Π, so that the first two requirements are fulfilled. Now note that letting $r = up + s$, u is odd if and only if r and s have opposite parities. Therefore, if s is set to a fixed odd value throughout the search sequence, it is

Alg. 3.2 Efficient-Cube-Root-Generation(ℓ_n)

Input: a bitsize ℓ_n, $\Pi = 3 \cdot 5 \cdot \ldots \cdot p_t$
Output: an ℓ_n-bit provable prime n

```
1. $\ell \leftarrow \ell_n$
2. while $\ell > 31$ do
3.     $\ell \leftarrow \lfloor \ell/3 \rfloor$
4. $\ell \leftarrow \ell + 1$
5. $n \leftarrow \mathsf{GenInitPrime}(\ell)$                    [compute the initial small prime]
6. while $\ell < \ell_n$ do
7.     $p \leftarrow n$
8.     $\ell \leftarrow \min(3\ell - 1, \ell_n)$
9.     $I \leftarrow \lfloor \frac{2^{\ell-1}}{2p} \rfloor$
10.    Select $r$ at random from $[I+1, 2I]$ such that $r = up + s$, $1 \leq s < p$ for odd $u$
       and $n \leftarrow 2rp + 1$ is coprime to $\Pi$ and go to 12
11.    Update $r$ in $[I+1, 2I]$ such that $r = up+s$, $1 \leq s < p$ for odd $u$ and $n \leftarrow 2rp+1$
       is coprime to $\Pi$
12.    if $\ell < 129$ then
13.        select $a$ at random from $[2, n-2]$
14.    else
15.        $a \leftarrow 2$
16.    if $a^{n-1} \bmod n \neq 1$ then go to 11
17.    if $\gcd(a^{2r} - 1, n) \neq 1$ then go to 11
18. return $n$
```

enough to ensure that r is even to force the parity of u to one. We now describe our method in more detail. Focusing on the search sequence associated with the i-th iteration, our generator proceeds as follows:

1. Fetch precomputed values $\Pi \leftarrow \Pi[i]$ and $\Lambda \leftarrow \Lambda[i]$ from code data. $\Pi \approx 2^{\ell_{i-1}-2}$ is a product of small odd primes (thereby excluding 2 from the factorization of Π), and Λ is the Carmichael function of Π.
2. Use [15] to generate a random invertible element $x \in \mathbb{Z}_{\Pi}^{\star}$, namely:
   (a) Randomly select x modulo Π
   (b) Compute $t = x^{\Lambda} \bmod \Pi$
   (c) If $t \neq 1$
      i. Randomly select z modulo Π
      ii. Update $x = x + z(1-t) \bmod \Pi$
      iii. Goto 2b
3. Compute $1/(2p) = (2p)^{\Lambda-1} \bmod \Pi$ and derive $1/p \bmod \Pi$
4. Randomly select an odd value s modulo p
5. Use Chinese remaindering to compute $r \in [0, 2p\Pi]$ such that $r = x - 1/(2p) \bmod \Pi$, $r = s \bmod p$ and $r = 0 \bmod 2$. More precisely:
   (a) Compute $r_{\Pi p} = (((x - 1/(2p) - s)/p) \bmod \Pi) \cdot p + s$
   (b) Compute $r = (r_{\Pi p} \bmod 2) \cdot \Pi \cdot p + r_{\Pi p}$
   (c) Add appropriate multiple of $2p\Pi$ to r to get $r \in [I+1, 2I]$

This concludes the initialization of the i-th loop *i.e.* the random selection of r at Step 10 at the i-th iteration. Updating r consists in just refreshing x as $x = 2x \bmod \Pi$ and performing a new round of Chinese remaindering as per Step 5 above. It is worthwhile noticing optimizations here: since p and s are fixed throughout the search sequence, the generator can just compute $1/p \bmod \Pi$ and $(-1/(2p) - s) \bmod \Pi$ once and for all and store these values. Step 5 then amounts to a couple of multiplications and additions. Also, modular exponentiations modulo Π are particularly efficient since Λ is small due to the particular form – extreme smoothness – of Π.

4 Estimating the Output Entropy

The rule for deriving at each iteration an ℓ_i-bit provable prime from an ℓ_{i-1}-bit other provable prime ($n \leftarrow 2rp + 1$) intrinsically generates primes p_i such that $p_i - 1$ is a multiple of a *half-size* prime p_{i-1}. This particular structure is not representative of the majority of prime integers, and obviously does not allow to generate them all. This section establishes the entropy of the output distribution of primes generated by Algorithms 3.1 and 3.2 [6] and compare the output entropy with that obtained by a perfect generator that outputs uniformly random primes of a given bitsize ℓ_n.

Let us denote by R_{ℓ_i} the number of ℓ_i-bit primes that are attainable by the Square Root method at the end of iteration i. Note that any one of them can be uniquely derived from the sequence $(r_1, \ldots, r_i)$ of the values taken by r at each iteration. Since r is drawn at random, this suggests the heuristic approximation that the distribution of generated primes is uniform and that its entropy is equal to $H_{\ell_i} = \log_2(R_{\ell_i})$. According to Gauss's theorem, the number $\pi(x)$ of primes lesser than x is well approximated by $\frac{x}{\ln(x)}$ for large x. The number of exactly ℓ-bit primes can thus be estimated by

$$S_\ell = \frac{2^\ell}{\ln(2^\ell)} - \frac{2^{\ell-1}}{\ln(2^{\ell-1})} \ .$$

In an initial step, the algorithm randomly generates an ℓ_0-bit prime p_0, so that $R_{\ell_0} = S_{\ell_0}$. For $x \in \left[2^{\ell_{i-1}-1}, 2^{\ell_{i-1}}\right]$, consider an interval of width dx centered on x. Every p_{i-1} in this interval can generate $I = \lfloor \frac{2^{\ell_i-1}}{2 \cdot p_{i-1}} \rfloor \simeq \frac{2^{\ell_i-2}}{x}$ candidates among which $\frac{2^{\ell_i-2}}{x \cdot \ln(2^{\ell_i})}$ are prime numbers[7]. The total number of primes – that can or cannot be reached by the generator – in the considered interval is $\frac{dx}{\ln(x)}$,

[6] Note that for efficiency purposes Algorithm 3.2 only selects r values for which $u = \lfloor \frac{r}{p} \rfloor$ is odd. In the sequel we first derive the entropy of our method when ignoring this trick. We subsequently address the effect of this feature later on.

[7] This derives from a commonly accepted approximation that the Chebotarëv density theorem also stands for large intervals. This theorem actually implies that for any coprime integers a and d, the proportion of primes less than x belonging to the arithmetic progression $\{a + nd\}_n$ tends to $\frac{1}{\phi(d)}$ when x tends to infinity.

but only a fraction

$$\frac{R_{\ell_{i-1}} \cdot \ln(2^{\ell_{i-1}})}{2^{\ell_{i-1}-1}}$$

of these can be generated at iteration $(i-1)$, so that the number of primes p_{i-1} to consider in the interval is

$$\frac{R_{\ell_{i-1}} \cdot \ln(2^{\ell_{i-1}}) \cdot dx}{2^{\ell_{i-1}-1} \cdot \ln(x)} \ .$$

Integrating over $\left[2^{\ell_{i-1}-1}, 2^{\ell_{i-1}}\right]$ the number of primes that each p_{i-1} can generate, we obtain

$$\begin{aligned}
\frac{R_{\ell_i}}{R_{\ell_{i-1}}} &\simeq \int_{2^{\ell_{i-1}-1}}^{2^{\ell_{i-1}}} \frac{\ln(2^{\ell_{i-1}}) \cdot 2^{\ell_i-2}}{2^{\ell_{i-1}-1} \cdot \ln(2^{\ell_i})} \cdot \frac{dx}{x\ln(x)} \\
&\simeq \frac{\ell_{i-1} \cdot 2^{\ell_i-2}}{\ell_i \cdot 2^{\ell_{i-1}-1}} \cdot \int_{2^{\ell_{i-1}-1}}^{2^{\ell_{i-1}}} \frac{dx}{x\ln x} \\
&\simeq \frac{\ell_{i-1}}{\ell_i} \cdot 2^{\ell_i-\ell_{i-1}-1} \cdot \big(\ln(\ell_{i-1}) - \ln(\ell_{i-1}-1)\big) \\
&\simeq \frac{\ell_{i-1}}{\ell_i} \cdot \frac{2^{\ell_i-\ell_{i-1}-1}}{\ell_{i-1}-1}
\end{aligned}$$

whence

$$R_{\ell_n} = S_{\ell_0} \cdot \frac{\ell_0}{\ell_n} \cdot \frac{2^{\ell_n-\ell_0-k}}{\prod_{i=1}^{k}(\ell_{i-1}-1)} \tag{3}$$

where examples cases for k, ℓ_0 and ℓ_i are given in Tables 1 and 2.

As mentioned above, Equation (3) does not take into account that only half of the values for r are selected as prime candidates in Algorithm 3.2. Assuming that even and odd values of u are evenly distributed for r ranging from $I+1$ to $2I$, the effect of ignoring half of potential candidates is that every prime p_{i-1} in the neighborhood of x can generate only $\frac{2^{\ell_i-3}}{x\cdot\ln(2^{\ell_i})}$ primes. This results in the following expression for the number of l_n-bit primes generated by Algorithm 3.2 when only odd u values are selected:

$$R_{\ell_n} = S_{\ell_0} \cdot \frac{\ell_0}{\ell_n} \cdot \frac{2^{\ell_n-\ell_0-2k}}{\prod_{i=1}^{k}(\ell_{i-1}-1)} \ . \tag{4}$$

The estimated entropies H_{ℓ_n} provided by Algorithms 3.1 and 3.2 are given in Table 3 for different output bitsizes ℓ_n together with the entropy $H^*_{\ell_n}$ of a perfectly uniform distribution.

The entropy loss of the proposed prime generation ranges from 36 bits for 512-bit primes to 57 bits for 2048-bit primes for the Square Root method, and only from 24 to 37 bits for the Cube Root method. While somewhat larger than the entropy loss of about 4 bits found in Maurer's method, it is noticeable that it is small enough so that exhaustive search remains infeasible for currently secure

ℓ_n	512	768	1024	1536	2048
$H^*_{\ell_n}$	503	758	1014	1525	2037
H_{ℓ_n} (Alg. 3.1, Eq. (3))	467	720	968	1476	1980
H_{ℓ_n} (Alg. 3.2, Eq. (4))	479	733	981	1490	2000

Table 3. Entropy loss w.r.t. ideal prime generation

bitsizes. We believe that the security of RSA and DSA cryptosystems is not (or only marginally) affected by using either Algorithm 3.1 or 3.2 for generating provable primes.

5 Implementation Results and Practical Aspects

5.1 On-board Generation of Probable Primes

Our implementation relies on an AT90SC chip supplied by Inside Secure embedding the Ad-X cryptoprocessor and the 8-bit AVR core both running at 30 MHz. The chip manufacturer provides a cryptographic toolbox for cryptography developers with all basic operations over large integers: modular multiplication, modular exponentiation, GCD, inversion, division, and so forth. The associated documentation provides estimated performances (cycle count) for these operations. Using this information we know the exact cycle count for any step of the generation algorithm. The exact average timings of our prime number generators can then be deduced on this component using Equation 1. Using the development kit from IAR running on a chip emulator loaded with the toolbox, the performance of our implementation of the generator for probable primes was experimentally confirmed to coincide perfectly with Equation 1.

The Fermat test with base 2 runs in 11 ms for a 512-bit integer n while the Miller-Rabin test with a random base is computed in 18 ms. We chose $t = 54$, so that Π is the product of small primes ranging from 2 to 251 and we choose $h = 3$ (the number of Miller-Rabin rounds).

On average, our generator outputs 512-bit probable primes in 580 ms ($N(512, \Pi) = 35.6$), 768-bit probable primes in $2'130$ ms ($N(768, \Pi) = 53.4$) and 1024-bit probable primes in $5'780$ ms ($N(1024, \Pi) = 71.2$).

5.2 Generating Provable Primes

Similarly, we deduced from Equation 2 the execution timings for our generator of provable primes on the same smartcard platform. We made use of the base-2 Fermat test when ℓ is greater than 128 bits, and took the same value for Π as in the case of probable primes. We have also implemented Algorithm 3.1 on the target chip. As a result, using the Square Root method to generate provable primes of respectively 512, 768 and 1024 bits requires on average 810, $2'580$ and $5'940$ ms. The Cube Root method decreases these figures to 760, $2'240$ and $5'700$ ms respectively.

5.3 Comparing Generators for Probable and Provable Primes

Given the expressions of $T_{\mathrm{Prob}}(\ell)$ and $T_{\mathrm{Provable}}(\ell)$, a rough guesstimate is that about the same number of modular exponentiations should be required to generate probable and provable primes of the same size, assuming trial divisions and identical values for Π. This is because the extra workload needed to generate the sequence of intermediate primes in the provable case remains fairly small compared to the resources needed to generate the full-length ℓ_n-bit provable prime. Moreover, this extra workload is somewhat compensated by the absence of final Miller-Rabin rounds or the Lucas test. All in all, we observe that the generation of a provable prime is slightly less efficient that the one of a probable prime when only a few Miller-Rabin rounds are required. However, the Cube Root algorithm becomes the fastest option when either a significant amount of Miller-Rabin iterations or a Lucas test is needed.

Figure 1 provides performance measurements for the various generation methods discussed in the paper.

Bitlength ℓ_n	h	512	768	1024	1536	2048	Lucas test
Algorithm 2.2	3	**640**	**2130**	**5780**	**25700**	**74400**	yes
Algorithm 2.2	40	1170	3700	9290	36800	98900	no
Algorithm 3.1	-	**810**	**2580**	**5940**	**26500**	**75600**	provable
Algorithm 3.2	-	**760**	**2240**	**5700**	**24400**	**73550**	provable

Fig. 1. Time (in milliseconds) measurements for various prime number generators.

We find that a Lucas test, as defined in FIPS 186-3, is roughly equivalent to 3.5 Miller-Rabin rounds and is therefore rather efficient on the AT90SC – comparatively to higher ratios found on other architectures. Overall, our experimental validation shows that the Cube Root method is essentially as efficient as the state-of-the-art generation algorithms for probable primes.

6 Achieving Leakage-Resistant Prime Number Generation

This section addresses side-channel attacks and ways to protect prime number generation from information leakage. Recent research works [11,8] have highlighted that prime number generation may be subject to power analysis. It is therefore necessary to ensure resistance against side-channels, especially when the device is operated in an untrusted environment. We give in this section a few guidelines for designing a protected implementation.

Assets to be protected are the output prime number as well as the secret elements used throughout its generation, more precisely the random values r and the sequence of intermediate primes reached by each iteration. It is therefore necessary to ensure that the implementation does not leak these values either during their generation or while they are being manipulated by the generation algorithm.

A first information leakage can occur during the generation of the first ℓ_0-bit prime. Since this is done using the Miller-Rabin criterion, the Miller-Rabin test itself has to be protected against side-channel attacks. A typical protection mechanism consists in performing an atomic modular exponentiation in the sense of [7] but since the base we use here is small, there is a risk that the exponent $n-1$ leaks at each multiplication as explained in [9]. The exponentiation may therefore be computed using a Square and Multiply-Always exponentiation which is a regular algorithm. A second operation to protect is the computation of I. This step involves the manipulation of p which must be kept secret. We therefore suggest to implement a secure division algorithm as described in [17].

Finke et al.presented in [11] an attack that specifically targets the computation of the next prime candidate (coprime to Π) at Step 2. of Algorithm 2.2. The attack is particularly applicable when a trial update operation is done with increments of 2 or Π. This attack does not seem applicable on Step 9 (performed with trial updates) of Algorithm 3.1 since the value used for next value of n is $n + 2p$ and p is unknown to the attacker. We recommend to implement the constructive method which is not sensitive to this attack and resists physical observation if the computation of p is done with the same exponentiation as the one used when applying the Miller-Rabin criterion.

We also note that the exponentiation $a^{n-1} \bmod n$ in Step 11 must be performed securely and that the atomic exponentiation is neither resistant nor efficient when $a = 2$. This part can be computed in a regular way using a Square and Multiply-Always exponentiation. In this case using $a = 2$ still results in negligible computational time for the multiplication and the computation remains protected against the SPA attack published in [9]. However the first squaring and multiplication operations (when the accumulator is still a power of 2 smaller than the modulus n) could leak information. It would then reveal the first bits of the exponent (about 10). It is then recommended to blind the modulus with a random value: in that case the computation would be $(2^{n-1} \bmod r_1 \cdot n) \bmod n$.

The final computations to protect from power analysis lie in Step 12. The exponentiation $2^{2r} \bmod n$ must be protected against the disclosure or r by using, as previously, the Square and Multiply-Always exponentiation technique. Also, the GCD operation $\gcd(2^{2r} - 1, n)$ could reveal the value of p if not implemented in a secure way. Our implementation of the GCD calculation has been carried out in constant time using dummy operations.

Applying these methods we obtain a side-channel protected efficient generator for provable primes. Finally, we note that fault-based attacks are not considered as a serious threat for prime number generators at the present time. This is mainly due to the inherently randomized nature of the generation algorithms.

7 Conclusion

The paper introduced two new methods to efficiently generate provable primes in embedded environments. We put forward novel algorithmic solutions and report practical results from our smartcard implementations. We have demonstrated that efficient generators exist for provable primes in constrained environments and compared the new methods with state-of-the-art generators for probable primes. We addressed side-channel analysis to ensure secure implementations of our generation methods. Overall, the paper opens the way to embedded generation of provable primes in nearly similar or better performances than current generators.

Acknowledgments

The authors would like to thank Vincent Verneuil for his valuable comments on this manuscript.

References

1. M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 2:781–793, 2002.
2. W. R. Alford, A. Granville, and C. Pomerance. On the difficulty of finding reliable witnesses. In *Proceedings of the First International Symposium on Algorithmic Number Theory*, pages 1–16, 1994.
3. N. C. Ankeny. The least quadratic non residue. *Annals of Mathematics*, 55:65–72, 1952.
4. A. O. L. Atkin and F. Morain. Elliptic Curves And Primality Proving. *Mathematics of Computation*, 61:29–68, 1993.
5. E. Bach. Explicit bounds for primality testing and related problems. *Mathematics of Computation*, 55:355–380, 1990.
6. J. Brillhart, D. H. Lehmer, J. L. Selfridge, B. Tuckerman, and Jr. S. S. Wagstaff. *Factorization of* $b^n \pm 1, b = 2,3,5,7,10,11,12$ *Up to High Powers*, volume 22. American Mathematical Society, 1988.
7. B. Chevallier-Mames, M. Ciet, and M. Joye. Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.
8. C. Clavier and J-S. Coron. On the Implementation of a Fast Prime Generation Algorithm. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 443–449. Springer, 2007.
9. J.-C. Courrege, B. Feix, and M. Roussellet. Simple Power Analysis on Exponentiation Revisited. In D. Gollman and J.-L. Lanet, editors, *Ninth Smart Card Research and Advanced Application IFIP Conference - CARDIS 2010*, volume 6035 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2010.
10. W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

11. T. Finke, M. Gebhardt, and W. Schindler. A New Side-Channel Attack on RSA Prime Generation. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2009.
12. FIPS PUB 186-3. *Digital Signature Standard.* National Institute of Standards and Technology, october 2009.
13. H. L. Garner. The residue number system. In *Proceedings of the Western Joint Computer Conference*, pages 146–153, 1959.
14. G. Jaechke. On strong pseudoprimes to several bases. *Mathematics of Computation*, 61:915–926, 1993.
15. M. Joye and P. Paillier. Fast generation of prime numbers on portable devices: An update. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 160–173. Springer, 2006.
16. M. Joye, P. Paillier, and S. Vaudenay. Efficient Generation of Prime Numbers. In Ç. K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000*, volume 1965 of *Lecture Notes in Computer Science*, pages 340–354. Springer, 2000.
17. M. Joye and K. Villegas. A protected division algorithm. In *Proceedings of the Fifth Smart Card Research and Advanced Application Conference, CARDIS '02*, 2002.
18. U. M. Maurer. Fast Generation of Secure RSA-Moduli with Almost Maximal Diversity. In *Advances in Cryptology - EUROCRYPT '89*, pages 636–647.
19. Ueli M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *J. Cryptology*, 8(3):123–155, 1995.
20. C. Pomerance, C. Selfridge, and J.L Wagstaff. The pseudoprimes to 25.10e9. *Mathematics of Computation*, 35:1003–1026, 1990.
21. R. L. Rivest, A Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM 21*, pages 120–126, 1978.
22. J. Shawe-Taylor. Generating strong primes. *Electronic Letters*, 22(16):875–877, 1986.

A Detailed Efficient Algorithms for our Method

Alg. A.1 Generation of the initial prime based on Miller-Rabin testing.

```
Input: bitsize $\ell_0 < 32$ of the initial (provable) prime, $\Pi = 2 \cdot 3 \cdot 5 \cdot \ldots \cdot p_t$
Output: $\mathsf{GenInitPrime}(\ell_0)$: a $\ell_0$-bit provable prime
1. generate a random $\ell_0$-bit integer $n$ with $\gcd(n, \Pi) = 1$ and go to 3
2. update $n$ such that $\gcd(n, \Pi) = 1$,
3. if $\mathsf{F}_2(n) =$ false then go to 2
4. if $\mathsf{MR}_2(n) =$ false then go to 2
5. if $\mathsf{MR}_7(n) =$ false then go to 2
6. if $\mathsf{MR}_{61}(n) =$ false then go to 2
7. return $n$
```