Recovering Private Keys Generated with Weak PRNGs

Pierre-Alain Fouque, Mehdi Tibouchi, Jean-Christophe Zapalowicz

Recovering Private Keys Generated With Weak PRNGs

Pierre-Alain Fouque[1], Mehdi Tibouchi[2], and Jean-Christophe Zapalowicz[3]

[1] Université de Rennes 1 and Institut universitaire de France
pierre-alain.fouque@ens.fr
[2] NTT Secure Platform Laboratories
tibouchi.mehdi@lab.ntt.co.jp
[3] Inria
jean-christophe.zapalowicz@inria.fr

Abstract. Suppose that the private key of discrete logarithm-based or factoring-based public-key primitive is obtained by concatenating the outputs of a linear congruential generator. How seriously is the scheme weakened as a result?

While linear congruential generators are cryptographically very weak "pseudorandom" number generators, the answer to that question is not immediately obvious, since an adversary in such a setting does not get to examine the outputs of the congruential generator directly, but can only obtain an implicit hint about them—namely the public key.

In this paper, we take a closer look at that problem, and show that, in most cases, an attack does exist to retrieve the key much faster than with a naive exhaustive search on the seed of the generator.

The problem is similar to the one considered by Bellare, Goldwasser and Micciancio regarding DSA and "pseudorandomness", and this line of work arguably has renewed relevance in view of the sensitive role that random number generation has been found to play in a number of recent noted papers, such as the one by Lenstra et al. at CRYPTO 2012.

Keywords: linear congruential generator, discrete logarithm, factoring, cryptanalysis

1 Introduction

Recently Lenstra et al. have proposed a *sanity check* of public keys collected on the web [20] and concluded that *generating secure public keys in the real world is challenging*. A related study by Heninger et al. [17] pointed in particular to the role of (pseudo)randomness generation as the chief cause for the weak keys observed in the wild. Cryptographers have to look at the security of pseudorandom generators used to generate the secret keys.

Pseudorandom generators are one of the main component of security products and their importance for security is hard to overestimate. A number of practical attacks stem from problems with randomness generators. Indeed, it is difficult to obtain randomness on computers and embedded devices, which tend to aggregate various more or less reliable sources of entropy (mouse movements, passwords, network interrupts, electronic noise) and use pseudorandom number generators to expand them into arbitrarily long, hopefully uniform-looking bit strings.

Such practical generators have been analyzed and a framework has been given by Barak and Halevi [1] for the Linux generators, who discuss the importance of the randomness collector which maintains a state of enough entropy and an extraction function whose aims is to output random bitstring from the state, which is then expanded into an arbitrarily long pseudorandom string. Cryptographers tend to concentrate on the second aspect: they assume that a pseudorandom generator is seeded with a uniformly distributed bitstring and the goal of the generator is to stretch the seed to a longer bitstring. They define security notions and a generator is called secure if it is hard to distinguish its output from uniformly distributed bitstring given the previous output bits.

The security of the first cryptographically secure generators has been based on some number-theoretic hard problems, in the sense that the problem of distinguishing the outputs from uniform is reduced to a number-theoretic problem. In 1981, Shamir proposed one generator based on the strong RSA problem in [25]; Blum and Micali proposed a generator based on the discrete logarithm problem [5]; Blum, Blum and Shub proposed a generator based on the factorization problem in [4]; and Micali and Schnorr defined another generator based on the problem of distinguishing small e-th root modulo a RSA modulus from uniform values in [23].

These generators are interesting from a theoretical point of view, but they are rather inefficient and in practice more efficient generators using symmetric cryptographic functions have been preferred. For instance, ISO and NIST propose generators using symmetric primitives such as hash functions and block ciphers, which are more efficient and can be modeled as random oracles, ideal ciphers, pseudorandom functions or permutations. In [14], Desai et al. proved the security of these symmetric primitive-based generators, provided that the underlying primitives satisfy certain standard assumptions.

Linear Congruential Generators. Cryptographers have also studied the security of the linear congruential generators (LCG), widely used in simulation.

These generators are efficient and have a very small memory footprint. Moreover, with suitable parameters, they can have good statistical properties such as a large period length and their output distribution is uniform. As they are also very easy to implement, they tend to be used in the standard libraries of many languages and compilers: the `rand` function proposed in the POSIX standard and the ones used in many C/C++ standard libraries, Java's `java.util.Random`, most implementations of `RAND` in Fortran, etc. Their efficiency and ubiquity make them attractive to implementors, especially in constrained environments, even in some security-sensitive applications.

Unfortunately, LCGs are cryptographically insecure: Boyar [9] proved that, with a sufficiently long run of the pseudorandom sequence, one can recover the seed in polynomial time in the size of the internal state and Stern [26] proved that this is also the case even if only the most significant bits of each successive states is revealed (see also [8,19,16]). These attacks are based on lattice reduction, usually LLL [21]. However, Contini and Shparlinski have proposed a careful analysis of these algorithms in [12] and established some limits to their applicability, indicating that with properly chosen parameters, the generator might become secure.

Related Work. In any case, these attacks assume that the adversary has direct access to a certain number of outputs (although the parameters of the generator may remain unknown). As a result, using such a cryptographically weak generator in a cryptographic protocol does not automatically make the resulting protocol insecure, because an adversary against the protocol may not have access to the actual outputs of the generator. This led Bellare et al. [2] to analyze the security of the Digital Signature Algorithm (DSA) when the random nonces used in signature generation are computed using a linear congruential generator. They showed that this does seriously break security: a few signatures are enough to recover the secret key. This started an important line of research on the security of DSA when partial information on the nonces is revealed. For instance, Smart and Howgrave-Graham [18] and later Nguyen and Shparlinski [24] showed that the knowledge of a small number of the most significant bits of the nonces allows to efficiently recover the secret key using LLL. Bleichenbacher even established [3] that the bias on the single most significant bit of the nonce that occurs when using some version of the NIST generator can be sufficient to efficiently recover the secret key.

Our contributions. In this paper, we investigate a question similar to the one considered by Bellare et al. but in a different direction than the DSA cryptanalysis papers. We consider the problem of the security of public-key schemes based on the hardness of the discrete logarithm problem or the factoring problem when the secret keys are constructed by concatenating the outputs of a linear congruential generator. Since the attacker does not get those outputs directly, but only an implicit hint, namely the corresponding public key, recovering the secret key is not trivial even if a cryptographically weak generator such as the

LCG is used. We show that this is usually enough to recover the secret key much faster than using the trivial exhaustive search on the seed of the generator.

Our attack relies on the assumption that the secret key is obtained as the concatenation of successive outputs of a linear congruential generator. We also assume as is usual in cryptography that the parameters of the LCG are public and therefore an exhaustive search on the seed allows to recover the secret key in time 2^k where k is the size of the seed. Typical parameters for the LCG are 32 bits or 64 bits internal state to allow fast implementation without using a library for large integer arithmetic. The main observation of our work is that if we split the seed of size k into two parts $(A \cdot 2^{k/2} + B)$ where A and B are $(k/2)$-bit long, the linearity of the LCG makes it "almost" possible to write the secret key as a sum $U + V \cdot 2^{k/2}$ where U (respectively V) only depends on B (resp. A) and the parameters of the LCG. This is correct up to carries, which do occur but can be taken care of separately. As a result, we can obtain a time-memory tradeoff on the search for the LCG seed when such generators are used to generate the secret key of a discrete logarithm-based scheme or to the prime factors of an RSA modulus. The discrete logarithm case is mainly a baby-step giant-step attack on the lower and upper halves of the seed, while the factoring case proceeds similarly using multipoint polynomial evaluation.

The main advantage of these attacks is that they allow key recovery from the public key alone, independently of any further information on the underlying cryptographic schemes.

Organization of the paper. The paper is organized as follows. After some preliminaries in §2, we present our attack in the discrete logarithm case in §3 and in the factoring case in §4. Finally, in §5, we give an overview of the complexity of our attacks for typical parameter sizes.

2 Preliminaries

We first recall the definition of the linear congruential generator and fix some notations which will be used in the following sections; then we briefly discussion multipoint evaluation of univariate polynomials, which will be used in the factoring case.

2.1 Linear congruential generator

For M an integer of size m bits, we denote by $\mathbb{Z}_M$ the ring of integers modulo M. The internal state of a linear congruential generator evolves according to the following recurrence relation:

$$v_{i+1} = a \cdot v_i + b \bmod M \tag{1}$$

where a and b are fixed constant in $\mathbb{Z}_M$ (the parameters of the generator) and $v_0 = s$ is the secret seed. The successive outputs o_i of the generator are the k least

(or most) significant bits v_i at each iteration (for some fixed $k \in \{1, \ldots, m\}$). Note that the recurrence equation is easily solved as:

$$v_i = a^i \cdot s + b \cdot (1 + a + \cdots + a^{i-1}) = a^i \cdot s + b_i \pmod{M}. \tag{2}$$

The following attacks rely on the assumption that a certain secret x is computed as a concatenation of successive outputs of such a linear congruential generator, with known parameters a, b and M. In other words, x can be written as:

$$x = o_0 + 2^k o_1 + \cdots + 2^{(r-1)k} o_{r-1} \tag{3}$$

for some fixed constant r. The secret x is then of size rk bits.

2.2 Multipoint evaluation of univariate polynomials

Let $P(x) \in \mathbb{Z}_N[x]$, with N an arbitrary integer, be a polynomial of degree less than $d = 2^k$. The multipoint evaluation problem is the task of evaluating P at d distinct points $\alpha_0, \ldots, \alpha_{d-1} \in \mathbb{Z}_N$. Using Horner's rule, it is easy to propose a solution that uses $O(d^2)$ additions and multiplications in $\mathbb{Z}_N$ but it is well-known that one can propose an algorithm with quasi-linear complexity $\widetilde{O}(d)$ operations in $\mathbb{Z}_N$ using a divide-and-conquer approach [15]; a better, more involved algorithm based on the middle product of polynomials has later been proposed in some special cases by Bostan and Schost [7,6]. That observation has found several applications in cryptanalysis [11,13].

A succinct description of the classical approach, based on product and remainder trees of polynomials is given in Appendix A. The complexity $T(d)$ of the recursive algorithm satisfies $T(d) = 2T(d/2) + O(M(d))$, where $M(i)$ denotes the arithmetic complexity to compute the product of two polynomials of degree i in $\mathbb{Z}_N[x]$, and therefore $T(d) = O(M(d) \log d)$.

3 The discrete logarithm case

We now consider key generation in a public-key scheme whose security is related to the discrete logarithm problem in some cyclic group $\mathbb{G}$ of prime order q and generator g. Typically, for such a scheme, $\mathbb{G}$, q and g are public parameters, the secret key contains a random element $x \in \mathbb{Z}_q$, and $h = g^x$ is revealed as part of the public key.

Assume that x is obtained from the outputs of a linear congruential generator of known parameters, as in Equation (1). The problem is to recover x from the public data faster than by an exhaustive search on the seed s.

Our approach in a nutshell is as follows. Separate the seed in its lower-order and higher-order halves: $s = u + 2^{k/2} \cdot v$ (we can assume for simplicity's sake that k is even). Then, by Equation (1), the internal state of the generator can be written as:

$$v_i = \left(a^i \cdot u + 2^{k/2} \cdot a^i \cdot v + b_i\right) \bmod M.$$

Thus, the corresponding output o_i can essentially be written, in turn, as the sum of a part depending only on u and another part depending only on v—only "essentially" because of possible carry bits and of possible overflows in the addition modulo M, but this can be taken care of, and we will ignore that for the moment.

Then, x is itself of the form $x = U + V$ where U is a publicly computable function of u, and V of v. In the group $\mathbb{G}$, this gives $h = g^U \cdot g^V$, or equivalently:

$$g^U = h \cdot g^{-V}.$$

Now, in time and space $O(2^{k/2})$, we can find a collision between the lists of elements of $\mathbb{G}$ of the form g^U for all $2^{k/2}$ possible values of u on the one hand, and $h \cdot g^{-V}$ for all $2^{k/2}$ possible values of v on the other hand, and hence recover the secret $x = U + V$.

The real algorithm has a slightly higher complexity due to the need to take carries and overflows into account, which we work out below first when $M = 2^k$ (the output is the full internal state) and then in the general case.

Note that since the parameters a and b are known, the constants b_i can be computed publicly and are thus irrelevant to the attack. To simplify notations, we can thus assume without loss of generality that $b = 0$.

Remark 1. The attack discussed here is generic and can of course be carried out in any cyclic group: it applies to (subgroups of) the multiplicative group of a finite field and to elliptic curves or abelian varieties alike. In the case of an elliptic curve group, the problem is to recover a secret value x from two points P, Q such that $Q = xP$, and when x is obtained from a linear congruential generator as before, it is again possible to divide x into two parts U and V, the first depending on u, the second on v. We can find a collision by checking an equality of the form $Q - UP = VP$.

3.1 Attack for non-truncated linear congruential generators

We first work out the details of this attack for a non-truncated linear congruential generator, which satisfies that $M = 2^k$. The non-truncated linear congruential generator is the most efficient of the linear congruential generator in the sense that it outputs the maximal number of available bits at each iteration.

Theorem 1. *Given two group elements $g, h \in \mathbb{G}$ with $h = g^x$, where x is an $(r \cdot k)$-bit exponent generated with a non-truncated linear congruential generator with public parameters and k-bit state, there exists an algorithm which retrieves the secret x in time and memory $O(2^{\frac{k+r}{2}})$.*

Proof. As mentioned previously, we may assume without loss of generality that the LCG has parameters such that $b = 0$. By Equation (2), its successive outputs are thus of the form:

$$o_i = v_i = (a^i \cdot s) \bmod M = (a^i \cdot s) \bmod 2^k.$$

Now write the seed as $s = u + 2^{k/2} \cdot v$, with u, v of $k/2$ bits. We get:

$$o_i = \left(a^i \cdot (u + 2^{k/2} \cdot v)\right) \bmod 2^k.$$

We can expand that expression for o_i using the following elementary lemma.

Lemma 1. *For all $\alpha, \beta, \gamma \in \mathbb{Z}$, $\gamma \neq 0$, there exists $\varepsilon \in \{0, 1\}$ such that:*

$$(\alpha + \beta) \bmod \gamma = (\alpha \bmod \gamma) + (\beta \bmod \gamma) - \varepsilon\gamma.$$

Proof. Indeed, let $L = (\alpha+\beta) \bmod \gamma$ and $R = (\alpha \bmod \gamma)+(\beta \bmod \gamma)$. Clearly, L and R are congruent modulo γ, so they must differ by a multiple of γ. Moreover, $0 \leq L < \gamma$ and $0 \leq R < 2\gamma$, hence $-\gamma < R - L < 2\gamma$, so $R - L$ must be of the form $\varepsilon \cdot \gamma$ with $\varepsilon \in \{0, 1\}$ as required. □

Thus, for all indexes i (and any choice of the two seed halves u, v), there exists $\varepsilon_i \in \{0, 1\}$ such that:

$$\begin{aligned} o_i &= (a^i \cdot u \bmod 2^k) + (a^i \cdot 2^{k/2} v \bmod 2^k) - \varepsilon_i \cdot 2^k \\ &= (a^i \cdot u) \bmod 2^k + 2^{k/2}(a^i \cdot v \bmod 2^{k/2}) - \varepsilon_i \cdot 2^k. \end{aligned}$$

If u and v are the two halves of the *correct* seed used to generate x, summing the $2^{ik} o_i$ yields, according to Equation (3):

$$x = U + V - Y$$

where:

$$\begin{aligned} U &= \sum_{i=0}^{r-1} 2^{ik} \cdot (a^i u \bmod 2^k) \\ V &= \sum_{i=0}^{r-1} 2^{ik+k/2} \cdot (a^i v \bmod 2^{k/2}) \\ Y &= \sum_{i=0}^{r-1} 2^{(i+1)k} \cdot \varepsilon_i. \end{aligned}$$

We can also decompose Y into a sum $W + Z$ where each of W and Z consist of $r/2$ terms, and obtain the relation $U - Z = x + W - V$, or equivalently:

$$g^{U-Z} = h \cdot g^{W-V}. \tag{4}$$

We can thus recover x by finding a collision between two lists of $2^{\frac{k+r}{2}}$ elements of $\mathbb{G}$, namely the g^{U-Z} (for all values of the half-seed u and all possible choices of the bits ε_i in Z) on the one hand, and the $h \cdot g^{W-Z}$ (for all values of the half-seed v and all possible choices of the bits ε_i in W) on the other. Using hash tables, this can be achieved in time and space $O(2^{\frac{k+r}{2}})$.

More precisely, one first computes the $2^{k/2}$ possible values U_i, the $2^{r/2}$ possible values Z_j and stores i, j in a hash table under the key $g^{U_i - Z_j}$. This table

contains $2^{\frac{k+r}{2}}$ different values accessible in constant time. Then one computes the $2^{k/2}$ possible values V_s, the $2^{r/2}$ possible values W_t and tests, for each of them, whether $h \cdot g^{W_t - V_s}$ is a key of the hash table. When the test succeeds, one obtains the correct values of u and v and can deduce the value x. The attack is summarized in Algorithm 1. □

Algorithm 1 Attack overview

Require: q, g, $h = g^x$, a, b, M
Ensure: x such as $h = g^x$
  Compute the hash table H by storing i, j at $H(g^{U_i - Z_j})$
  for each (V_s, W_t) **do**
    if $H(h \cdot g^{W_t - V_s})$ exists **then**
      return $x \leftarrow U_i + V_s - Z_j - W_t$
    end if
  end for

3.2 Attack for truncated linear congruential generators

We now consider a truncated linear congruential generator with a modulus $M > 2^k$ of size m (with $m < rk$). It is typically less efficient than the non-truncated one, but may have better properties in statistical and security terms. The attack we obtain has a slightly worse complexity than in the non-truncated setting.

Theorem 2. *Given two group elements $g, h \in \mathbb{G}$ with $h = g^x$, where x is an $(r \cdot k)$-bit exponent generated with a truncated linear congruential generator outputting the k most (or least) significant bits at each iteration, with public parameters and m-bit state, there exists an algorithm which retrieves the secret x in time and memory $O(2^{m/2} \cdot 5^{r/2})$.*

Proof. The principle of the attack remains similar however the carry is in a larger set of values. As a consequence the complexity in time and in memory is increased. Indeed, starting from Equation (2) with $b = 0$, the successive outputs are now of the form:

$$o_i = v_i \bmod 2^k = \big((a^i \cdot s) \bmod M\big) \bmod 2^k$$

in the case where the least significant bits are output, and:

$$o_i = v_i \gg (m-k) = \big((a^i \cdot s) \bmod M\big) \gg (m-k)$$

in the most significant bits case. By writing s as $u + 2^{m/2} v$, we get:

$$v_i = \big(a^i \cdot (u + 2^{m/2} \cdot v)\big) \bmod M.$$

and Lemma 1 ensures that:

$$(a^i \cdot u) \bmod M + 2^{m/2}(a^i \cdot v \bmod M) = v_i \text{ or } v_i + M.$$

Using the same lemma and the fact that $o_i = v_i \bmod 2^k$ (LSB case), we obtain:

$$(a^i \cdot u \bmod M) \bmod 2^k + (2^{m/2} \cdot a^i \cdot v \bmod M) \bmod 2^k = \begin{cases} o_i \\ o_i + 2^k \\ o_i + (M \bmod 2^k) \\ o_i + (M \bmod 2^k) + 2^k \\ o_i + (M \bmod 2^k) - 2^k \end{cases}$$

In the MSB case, we have (by denoting $j = m - k$) $o_i = v_i \gg j$ and thus:

$$(a^i \cdot u \bmod M) \gg j + (2^{m/2} \cdot a^i \cdot v \bmod M) \gg j = \begin{cases} o_i \\ o_i - 1 \\ o_i + (M \gg j) \\ o_i + (M \gg j) + 1 \\ o_i + (M \gg j) - 1 \end{cases}$$

Therefore, by applying the attack as before and using Equation (4), we have to find a collision between two sets of $2^{m/2} \cdot 5^{r/2}$, the factor $2^{m/2}$ coming from the search of U (respectively V) and the factor $5^{r/2}$ coming from the search of Z (respectively W). □

4 The factoring case

The attacks extend to public-key schemes whose security is related to the hardness of factoring, or of the RSA problem.

Denoting p and q two secret primes obtained from outputs of a linear congruential generator, and N the resulting product published as part of the public key, we would like to find p and q given N and the parameters of the generator.

The idea is again to separate the seed into a lower-order and a higher-order part, and to obtain a time-memory tradeoff compared to exhaustive search. The key ingredient is multipoint polynomial evaluation.

4.1 Basic prime generation

We first consider a prime number generation algorithm (see [22]) which consists in, from a random seed, computing the required number of outputs, concatenating them and using a probabilistic primality test such as Miller-Rabin or a deterministic one such as the AKS primality test. If the test fails, one selects another random seed and restarts the algorithm: all primality tests are independent.

As before, we consider the case where the linear congruential generator is not truncated and the case where it is truncated.

Theorem 3. *Given a RSA modulus N with $N = pq$, where p is an $(r \cdot k)$-bit prime generated with a non-truncated linear congruential generator (resp. a truncated linear congruential generator outputting the k most or least significant bits at each iteration), with public parameters and k-bit state (resp. m-bit state), there exists an algorithm which factorizes N in time and memory $\widetilde{O}(2^{\frac{k+r}{2}})$ (resp. $\widetilde{O}(2^{m/2} \cdot 5^{r/2})$) with overwhelming probability.*

Proof. For simplicity, we will treat the case of the non-truncated LCG since the use of a truncated one implies only a difference in the exhaustive search of the carry. By splitting the seed as $s = u + 2^{k/2} \cdot v$, we can write p as $p = U + V - Y$ with:

$$U = \sum_{i=0}^{r-1} 2^{ik}(a^i u \bmod 2^k)$$
$$V = \sum_{i=0}^{r-1} 2^{ik}(2^{k/2}(a^i v \bmod 2^{k/2}))$$
$$Y = \sum_{i=0}^{r-1} 2^{(i+1)k} \cdot \varepsilon_i.$$

We can also cut Y into two $r/2$-bit elements W, Z. Let us denote $A = U - Z$ and $B = V - W$ and suppose having $c(A+B) \bmod N = cp \bmod N$ with c an integer. Then, except the rare case where c is a multiple of q, this value is necessarily a multiple of p. Indeed $cp \bmod pq$ can only have the values 0 (case where $q|c$), $p, \cdots, (q-1)p$. Thus, a greater common divisor computation (GCD) with N will reveal p.

As an attacker, one does not have access to the correct value of the seed. Since u and v can take $2^{k/2}$ distinct values, one can compute the same amount of values U and V. Moreover there are $2^{r/2}$ possibilities for W and Z. In other words, the values $A = U - Z$ and $B = V - W$ are in two sets of $2^{\frac{k+r}{2}}$ elements and we have to find a test in order to determine the good ones.

More precisely, one first computes the $2^{\frac{k+r}{2}}$ different values $B_{s,t}$ by generating the values V_s and W_t and we consider the following polynomial of degree $2^{\frac{k+r}{2}}$:

$$P(X) = \prod_{s,t}(X + B_{s,t}) \bmod N$$

Then one computes the $2^{\frac{k+r}{2}}$ possible values $A_{i,j}$ by generating the values U_i and Z_j and proceeds a multi-evaluation of the polynomial P at the points $A_{i,j}$. The result is a set of $2^{\frac{k+r}{2}}$ values of the form:

$$\Big\{ \prod_{s,t}(A_{i,j} + B_{s,t}) \bmod N \quad | \quad i = 0, \cdots, 2^{k/2} - 1, j = 0, \cdots, 2^{r/2} - 1 \Big\}.$$

Finally one has to compute a test to detect the correct values A and B. It is done by computing a GCD between each value $(A_{i,j})$ and the public modulus

N. Since all the values of the seed and all the values of the carry are efficiently tested, the prime p will be recovered except if $P(A)$ is equal to 0. However this failure case is extremely rare: it requires that at least one of the $d-1$ integers composing the product with p is the prime q. The probability is thus equal to $\frac{d-1}{2^{\log q}}$. The attack is summarized in Algorithm 2. □

Algorithm 2 Attack overview (case $M = 2^k$)

Require: $N = pq$, a, b, M
Ensure: p such as $N = pq$
  Generate the polynomial $P(X) = \prod_{s,t}(X + V_s - W_t) \bmod N$
  Multi-evaluate P at the points $A_{i,j} = U_i - Z_j$
  for each point $A_{i,j}$ **do**
    if $\gcd(P(A_{i,j}), N) \neq 1$ **then**
      return $\gcd(P(A_{i,j}), N)$
    end if
  end for

4.2 Improved prime generation

Since there is no link between each probabilistic primality test in the first prime number generating algorithm, the failed tests are useless and free of cost from the point of view of the attacker. We now propose another one with a link by using a counter.

PRIMEINC Method. The PRIMEINC algorithm is a prime number generating algorithm proposed by Brandt and Damgård in [10] which basically picks a random number and increases it until a prime is found. In other words, if p is not a prime (but odd), then $p = p + 2$ and repeat. According to the prime number theorem, we expect to find a prime number after $\log p$ trials on average.

Corollary 1. *Considering the two cases of Theorem 3 coupled with the PRIMEINC algorithm, there exists an algorithm which factorizes N in time and memory $\widetilde{O}(2^{\frac{k+r}{2}})$ (resp. $\widetilde{O}(2^{m/2} \cdot 5^{r/2})$) with overwhelming probability.*

Proof. In our attack, we now search the correct value of p such as $p = A + B + 2\epsilon$ with $\epsilon \in \{0, \cdots, \log p\}$ (see Remark 2 for the size of the set). Thus, after the multi-evaluation, our algorithm should have computed a set covering the entire space of search as follows:

$$\Big\{ \prod_{s,t}(A_{i,j} + B_{s,t} + 2\gamma) \bmod N \quad | \quad i \leq 2^{k/2} - 1, j \leq 2^{r/2} - 1, \gamma \leq \log p \Big\}.$$

An efficient way to compute the search of the correct value of γ (i.e. $\gamma = \epsilon$) consists in applying the same trick as before, i.e. writing γ as $\gamma = \gamma_{MSB} + \gamma_{LSB}$

by splitting the bits into two parts.
In other words, in the first part of the algorithm, one computes the different values $B_{s,t}$ and the $\sqrt{\log p}$ values of γ_{LSB}. In the second part, one focus on the different values $A_{i,j}$ and the $\sqrt{\log p}$ values of γ_{MSB}. Thus, after the multi-evaluation, the resulted set corresponds to $2^{\frac{k+r}{2}}\sqrt{\log p}$ values of the form (case $M = 2^k$):

$$\prod_{s,t,\gamma}(A_{i,j} + B_{s,t} + 2(\gamma_{MSB} + \gamma_{LSB})) \bmod N.$$

With overwhelming probability, the value containing p does not contain q too and the test using the greatest common divisor will reveal p.

The modification due to the PRIMEINC method thus increases the complexity in time and in memory by a factor of $\sqrt{\log p}$, which disappears in the $\widetilde{O}$ since $\sqrt{\log p} = O(\sqrt{rk}) = O\big(\frac{r+k}{2}\big)$. □

Remark 2. Brandt and Damgård prove in [10] that the failure is about equal to $e^{-2\ell}$ when applying $\ell \cdot \log p$ iterations of the PRIMEINC algorithm. In the proof, we put $\ell = 1$, leading to a success rate of 86%.

5 Complexity estimates for concrete parameter sizes

Table 1 below presents the time complexity of our attack in the discrete logarithm case for typical parameter LCG sizes, as found in implementation of the `random` functions of common compilers and standard libraries, namely a modulus equal to either 2^{32} or 2^{64} (so that modular addition and multiplication can be implemented as simple operations on standard size registers), and an output size equal to either the full modulus size or half of it (corresponding to the top or bottom half of the state). The complexities are to be compared with that of the trivial attack: exhaustive search on seed.

Remark 3. Note that the differences of complexity between a linear congruential generator which outputs the least significant bits or the most significant bits is due to the fact that there are only three possibilities for the value of $(a^i \cdot u \bmod M) \bmod 2^k + (2^{m/2} \cdot a^i \cdot v \bmod M) \bmod 2^k$. Indeed, taking $M = 2^{32}$ or $M = 2^{64}$ yields $M \bmod 2^k = 0$.

Remark 4. In a few cases, for 16-bit output size, the complexity is in fact worse than the exhaustive search on the seed. This happens in the truncated case only, when r (the number of LCG outputs used to construct the secret) is particularly large, namely when $5^{r/2}$ (MSB case), resp. $3^{r/2}$ (LSB case), is greater than $2^{m/2}$.

Remark 5. In the factoring case, the complexities are larger by a logarithmic factor, from the use of quasilinear multipoint polynomial evaluation.

Secret size	Modulus	Output size	Attack complexity	
160	2^{32}	32	$2^{18.5}$	
160	2^{32}	16	$2^{23.9}$	$2^{27.7}$
160	2^{64}	64	$2^{33.5}$	
160	2^{64}	32	2^{36}	$2^{37.8}$
256	2^{32}	32	2^{20}	
256	2^{32}	16	$2^{28.7}$	$2^{34.6}$
256	2^{64}	64	2^{34}	
256	2^{64}	32	$2^{38.3}$	$2^{41.3}$
512	2^{32}	32	2^{24}	
512	2^{32}	16	$2^{41.4}$	$2^{53.2}$
512	2^{64}	64	2^{36}	
512	2^{64}	32	$2^{44.7}$	$2^{50.6}$
1024	2^{64}	64	2^{40}	
2048	2^{64}	64	2^{48}	

Table 1. Overview of our Attacks complexities in the discrete logarithm case. When the output size is smaller than the modulus, the first number corresponds to the LSB case, and the second one to the MSB case.

6 Conclusion

In this paper, we present new key-recovery attacks on discrete logarithm and factoring-based public-key schemes whose private keys are generated by concatenating the outputs of a linear congruential generator. Even though the LCG itself is known to be a cryptographically weak pseudorandom generator, it is not *a priori* obvious that its use would make key generation insecure, as its outputs are never revealed in clear to an adversary. It turns out, however, that the implicit hint about those outputs provided by the public key is enough to recover the private key significantly faster than with an exhaustive search on the seed.

It is hoped that this attack can be generalized to other, less naive pseudorandom generators in further work. Moreover, even in the case of the LCG, it would be interesting to extend it to different scenarios, such as the generation of randomness used in padding functions for encryption and signatures, or to settings where LCG parameters are unknown to the attacker.

References

1. Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to `/dev/random`. In *ACM CCS*, pages 203–212, 2005.
2. Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. "Pseudo-random" number generation within cryptographic algorithms: The DDS case. In *CRYPTO*, pages 277–291, 1997.

3. Daniel Bleichenbacher. On the generation of one-time keys in DL signature schemes. Presentation at the IEEE P1363 Working Group meeting, November 2000.
4. Lenore Blum, Manuel Blum, and Mike Shub. A simple unpredictable pseudo-random number generator. *SIAM J. Comput.*, 15(2):364–383, 1986.
5. Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, 1984.
6. Alin Bostan and Éric Schost. On the complexities of multipoint evaluation and interpolation. *Theor. Comput. Sci.*, 329(1-3):223–235, 2004.
7. Alin Bostan and Éric Schost. Polynomial evaluation and interpolation on special sets of points. *J. Complexity*, 21(4):420–446, 2005.
8. Joan Boyar. Inferring sequences produced by a linear congruential generator missing low-order bits. *J. Cryptology*, 1(3):177–184, 1989.
9. Joan Boyar. Inferring sequences produced by pseudo-random number generators. *J. ACM*, 36(1):129–141, 1989.
10. Jørgen Brandt and Ivan Damgård. On generation of probable primes by incremental search. In *CRYPTO*, pages 358–370, 1992.
11. Yuanmi Chen and Phong Q. Nguyen. Faster algorithms for approximate common divisors: Breaking fully-homomorphic-encryption challenges over the integers. In *EUROCRYPT*, pages 502–519, 2012.
12. Scott Contini and Igor Shparlinski. On Stern's attack against secret truncated linear congruential generators. In *ACISP*, pages 52–60, 2005.
13. Jean-Sébastien Coron, Antoine Joux, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Cryptanalysis of the RSA subgroup assumption from TCC 2005. In *PKC*, pages 147–155, 2011.
14. Anand Desai, Alejandro Hevia, and Yiqun Lisa Yin. A practice-oriented treatment of pseudorandom number generators. In *EUROCRYPT*, pages 368–383, 2002.
15. Charles M. Fiduccia. Polynomial evaluation via the division algorithm: The fast fourier transform revisited. In *STOC*, pages 88–93, 1972.
16. Alan M. Frieze, Johan Håstad, Ravi Kannan, Jeffrey C. Lagarias, and Adi Shamir. Reconstructing truncated integer variables satisfying linear congruences. *SIAM J. Comput.*, 17(2):262–280, 1988.
17. Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security '12*, 2012.
18. Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptography*, 23(3):283–290, 2001.
19. Antoine Joux and Jacques Stern. Lattice reduction: A toolbox for the cryptanalyst. *J. Cryptology*, 11(3):161–185, 1998.
20. Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Public keys. In *CRYPTO*, pages 626–642, 2012.
21. Arjen K. Lenstra, Hendrik W. Lenstra Jr., and László Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261(4):515–534, 1982.
22. Alfred Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. 1996.
23. Silvio Micali and Claus-Peter Schnorr. Efficient, perfect polynomial random number generators. *J. Cryptology*, 3(3):157–172, 1991.
24. Phong Q. Nguyen and Igor Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *J. Cryptology*, 15(3):151–176, 2002.

25. Adi Shamir. The generation of cryptographically strong pseudo-random sequences. In Allen Gersho, editor, *CRYPTO*, page 1. U. C. Santa Barbara, Dept. of Elec. and Computer Eng., ECE Report No 82-04, 1981.
26. Jacques Stern. Secret linear congruential generators are not cryptographically secure. In *FOCS*, pages 421–426. IEEE Computer Society, 1987.

A Multipoint evaluation of univariate polynomials

Here is a succinct description of the classical approach, based on product and remainder trees of polynomials. Let $P_0 = \prod_{\ell=0}^{d/2-1}(x-\alpha_\ell)$ and $P_1 = \prod_{\ell=d/2}^{d-1}(x-\alpha_\ell)$ and let us define $R_0 = P \bmod P_0$ and $R_1 = P \bmod P_1$. We have $R_0(\alpha_i) = P(\alpha_i)$ for all $i \in \{0,\ldots,d/2-1\}$ and $R_1(\alpha_i) = P(\alpha_i)$ for all $i \in \{d/2,\ldots,d-1\}$ and this gives immediately a recursive algorithm (i.e. compute P_0, P_1, R_0, R_1 and reduce the problem to the multipoint evaluation of R_0 and R_1 of degree $d/2 = 2^{k-1}$).

Let $A_i(x) = (x-\alpha_i)$ for $i \in \{0,\ldots,d-1\}$ and $P_{i,j} = A_{j2^i}A_{j2^i+1}\ldots A_{j2^i+2^i-1}$ for $i \in \{0,\ldots,k\}$ and $0 \le j < 2^{k-i}$. We have $P_{0,j} = A_j$ and $P_{i+1,j} = P_{i,2j}P_{i,2j+1}$ so for $i \in \{0,\ldots,k\}$ we can compute recursively all polynomials $P_{i,j}$ and $0 \le j < 2^{k-i}$ in $2^{k-i-1}O(M(2^i)) = O(M(d))$ operations in $\mathbb{Z}_N$ where $M(i)$ denotes the arithmetic complexity to compute the product of two polynomials of degree i in $\mathbb{Z}_N[x]$. Overall, the computation of all polynomials $P_{i,j}$ requires $O(M(d)\log d)$ operations in $\mathbb{Z}_N$.

The polynomials R_0 and R_1 can be computed using $O(M(d))$ operations in $\mathbb{Z}_N$ (using a Newton inversion), hence the complexity $T(d)$ of the recursive algorithm satisfies $T(d) = 2T(d/2) + O(M(d))$ and therefore $T(d) = O(M(d)\log d)$.