Cryptographic Insecurity of the Test&Repeat Paradigm

Tomáš ROSA
eBanka, a.s., Václavské náměstí 43
109 00 Prague 1, Czech Republic, EU, trosa@ebanka.cz

Abstract. Let $f(x)$ be a certain cryptographic function and let g, g: $Im(f) \rightarrow$ {true, false}, be an integrity test saying whether a particular value of $f(x)$ fits into predefined integrity boundaries or not. The "Test and Repeat" paradigm is then characterized by the following pseudocode: *repeat let* $y = f(x)$ *until* $g(y) = true$. On a first look, it may seem like a kind of best programming practice that can only improve overall security of the module. Especially, an architect can see it as a rather strong countermeasure against attacks based on computational faults – so called fault attacks. In this article, however, we will show that such a practice can induce particular cryptographic weaknesses. Therefore, it cannot be regarded as a general security improvement. Especially, it can even increase a vulnerability to the fault attacks. Its usage in cryptographic modules shall, therefore, undergo a proper cryptanalysis before being actually deployed.

Keywords. Fault attack, Side channel, Covert channel, DSA, Vernam cipher, RNG, Lattice, Cryptanalysis.

Introduction

In practice, we can meet the "Test and Repeat" (TAR) paradigm in many software and hardware applications, including cryptographic modules. Roughly speaking, it is a technical construction that encapsulates the following two very basic demands of computing systems architects:

1. To prevent a propagation of faulty results.
2. To ensure certain level of robustness of the application being designed.

The first aim is achieved by using the test part of TAR, while the second one is achieved by repeating the computation together with the test part several times before giving up the whole operation. Arranged in this way, TAR might also be regarded as a rather strong countermeasure against fault attacks ([1], [2]), especially against those ones based on an analysis of faulty results of a corrupted computation.

This paper was written as a tutorial note of a cryptologist to security engineers who design the hardware and-or software architecture of cryptographic modules. We will see two illustrative examples serving as a proof that TAR shall not be regarded as a general countermeasure, since there are realistic attacks that can be even right allowed or, at least, accelerated and hidden by it. The first example in §1 concerns a subliminal covert channel [2] occurring in Vernam's one-time pad [3] equipped with a random number generator [3] which uses TAR. The second example in §2 discusses a fault attack presented in [4]. This attack is focused on DSA and we will see that it can pass totally undetected when a "reasonable" TAR is applied. Moreover, TAR actually helps an attacker to keep her attack hidden and unrecognized here.

1. Case 1: TARed RNG

Let us imagine the following scenario that is based on real situations arising in the military area: Alice works with highly secured classified information which Bob wants to see. However, despite working for the same company, Bob does not possess the required clearance level. So, he seduces Alice and convinces her to cooperate with him on the classified data theft. To prevent such attacks, there are often strict technical countermeasures applied, so Alice cannot simply copy the clear data on a CD ROM and pass them to Bob. She can only do some operations with the clear data within her terminal. Any data written to any removable media or sent through a network are encrypted. An undesirable way allowing Alice to inconspicuously send the clear data to Bob is then referred to as a covert channel or even as a subliminal covert channel ([2], [5], [6]), depending on a technique used to create the channel. Obviously, a considerable effort is made to eliminate all these channels [6].

Now, let us assume that for encrypting the networked data from Alice, the system uses the Vernam's one-time pad cipher [3]. Denoting the plaintext data from Alice as an N-bit binary vector $M = (m_1, m_2, \ldots, m_N)$, the corresponding ciphertext $C = (c_1, c_2, \ldots, c_N)$ is computed as:

$$C = M \oplus K, \text{ i.e. } c_i = m_i \oplus k_i, \text{ for } 1 \leq i \leq N, \quad (1)$$

where $K = (k_1, k_2, \ldots, k_N)$ is a keystream of the same length as the message being encrypted. The keystream is a-priori unknown for both Alice and Bob. Arranged this way, the scheme is close to being unconditionally secure, i.e., for instance, unbreakable regardless an attacker computing power. A necessary condition is, however, that the keystream bits are independent and uniformly distributed random binary variables. Therefore, the scheme needs what is usually called a cryptographically strong random number generator (RNG). Note that in practice, such a generator would probably be based on a physical source of randomness (diode noise, etc.) which needs to be checked periodically for a malfunction (manifesting itself as a statistical irregularity in output data). Such a testing shall, however, not be arranged as TAR: If a statistical singularity occurs, the device must be put out of working order. Note that such a singularity does not imply that the device is really corrupted. Actually, a lot of alarms will be false. Therefore, in practice, it may be tempting to design a "cost saving" RNG which will automatically restart after the alarm giving the hardware next chance to pass the test. Unfortunately, devices with this behavior were already met in practice.

Provided that a TARed RNG is used for the keystream computation and Bob has an access to the ciphertext C, the construction of a subliminal covert channel from Alice to Bob is easy: Let us denote the statistical test applied on each L-bit binary block $B = (b_1, b_2, \ldots, b_L)$ produced by this RNG as g: $\{0, 1\}^L \rightarrow \{\text{true}, \text{false}\}$. If

$g(B)$ = true, then the block passed the test. Otherwise, the RNG generates a new block and repeats the test. For the sake of simplicity, let $L \mid N$ and assume the keystream is constructed from N/L blocks as $K = B_1 \parallel B_2 \parallel \ldots \parallel B_{N/L}$. Then there is no i, $1 \le i \le N/L$, such that $g(\,(k_{L(i-1)+1}, k_{L(i-1)+2}, \ldots, k_{Li})\,)$ = false, since all the blocks producing an alarm were filtered out by TAR. If Alice sends a plaintext message M consisting of N zero bits, we get C satisfying:

$$g(\,(c_{L(i-1)+1}, c_{L(i-1)+2}, \ldots, c_{Li})\,) = \text{true, for all } 1 \le i \le N/L, \quad (2)$$

since $M = 0$ implies $C = K$ according to Eq. (1). On the other hand, if Alice encrypts a block of N uniformly distributed independent random bits, then for all $1 \le i \le N/L$, there is a nonzero probability p that $g(\,(c_{L(i-1)+1}, c_{L(i-1)+2}, \ldots, c_{Li})\,)$ = false. The value of p corresponds with the probability of the false alarm for the particular test being used. Therefore, observing C long enough, Bob can distinguish if Alice encrypted a zero message or a message of random bits. From here, Bob can gain 1 bit of information. He gets another bit from another transmission, and so on. The expected length T of a ciphertext needed for 1 bit transmission can be estimated as $T = Lp^{-1}$. For example, let the testing function g implement the continuous RNG test defined in §4.9.2 of FIPS 140-2 [7] with the block size L = 16 b. This test returns false if and only if the two consecutive blocks of 16 bits produced by the tested RNG are the same. Therefore, the probability of a false alarm is $p = 2^{-16}$ and $T = 16{*}2^{16} = 2^{20}$. So, Bob needs to observe approx. 1 Mb of ciphertext for gain of 1 bit of secret information from Alice. We see, that the channel can be hardly used for a transmission of common data files, nevertheless, it may suffice for revealing a secret password, safe lock combination, etc. Moreover, Alice and Bob can use error control codes to increase a reliability of their covert channel. We can also observe that for a certain types of messages, the covert channel discussed above can spontaneously convert to a side channel allowing an attacker to gain some secret information even without cooperation with the sender.

1.1. A Cautionary Note

Although we may reasonably assume that a professional cryptographer should avoid designing the aforesaid illustrative scheme, it is worth noting that the problem can be more complicated. Let us assume that an RNG is put of working order immediately as the test says false. However, the service of a keystream generation must remain available. To maintain required availability, several backup RNGs may be installed. Each of them starts when its ancestor stops. The problem with covert channel might seem to be solved, but it is not. If all of these RNGs use the same statistical tests, then we still know that $M = 0$ implies Eq. (2), while for a random M, there can be false alarms detected over the ciphertext C. Therefore, a communication from Alice to Bob is still allowed. We see that the problem of reasonable fault detection versus covert channels minimization deserves closer attention, which is, however, beyond the scope of this paper.

2. Case 2: TARed DSA

It is well known cryptanalytical result that the security of a DSA [8] private key strongly depends on statistical properties of temporary nonces (i.e. Numbers-used-ONCE; usually denoted as k) used for a particular signature generation. Such a nonce must have a uniform distribution on a certain interval and must be kept secret. Otherwise, an undesirable subliminal side channel is created that enables a consecutive leakage of the private key information in every signature made. Having collected enough such signatures, an attacker can recover the whole private key with a trivial complexity on a general personal computer or a notebook. In 2002, Nguyen and Shparlinski presented a theoretically stable and practically very fruitful approach [9] to private key recovering which employs a lattice-based solution of Nguyen's variant of the hidden number problem (HNP) introduced in 1996 by Boneh and Venkatesan [10]. They results show, for instance, that we can recover the whole private key knowing only as few as the lowest three bits of each nonce for only 100 signatures made.

In 2004, the paper of Naccache, Nguyen, Tunstall, and Whelan was made public on IACR's ePrint (eprint.iacr.org) [11]. It connects the results obtained by Nguyen and Shparlinski together with a vulnerability to a fault injection observed for a certain kind of smartcard. Naccache et al. demonstrated that it was possible to use the fault injection to substitute known values for the lowest bytes of each nonce k. Besides the others, it is interesting to observe that this is such a kind of fault attack that cannot be prevented simply by checking each signature for faults by verifying its validity using a public key. Obviously, every signature made in this way is valid. So, despite seeming robust on a first look, it turns out that a countermeasure based on TAR is totally useless against this attack. The article [4] went further this way. It presented such a lattice-based fault attack on the DSA scheme that becomes even more dangerous when the device under attack behaves according to a "reasonable" TAR scenario: Using the public key, the device checks every signature made whether it is valid or not. Only valid signatures can be read from the device – this constitutes the test part of TAR. Furthermore, the module restarts the signing procedure automatically, until a valid signature is computed or the number of attempts is out of a predefined boundary – this is the repeat part of TAR. Since designers of such a module would probably require certain level of robustness and independence, we may reasonably assume that the device would allow even hundreds of repetitions before it blocks.

2.1. Implicit Verification of DSA Signatures

Let (p, q, g) denote DSA public parameters according to [8]: p, q are primes, such that $2^{1023} < p < 2^{1024}$, $2^{159} < q < 2^{160}$, $q \mid p - 1$, and g is a generator of a cyclic multiplicative subgroup $\boldsymbol{G}$ of $\boldsymbol{Z}_p^*$ of order $|\,\boldsymbol{G}\,| = q$. Furthermore, let x be a private key, $x \in \boldsymbol{Z}$, $0 < x < q$, and let y be a public key, $y = g^x \bmod p$. We assume that a cryptographic module employing TAR paradigm (possibly as a countermeasure against fault attacks) would behave according to the following algorithm. The notation of input parameters respects the fact that, in practice, the public parameters are usually stored independently with both records of the public and the private key.

Algorithm 1. Signing a message using DSA with implicit verification.
Input: Message to be signed m, private key record (p, q, g, x), public key record (p, q, g, y), repeat boundary B.
Output: Signature (r, s) or FAILURE.
Computation:

```
1. Let $i = 1$.
2. Choose an integer nonce $k$ at random, such that $0 < k < q$.
3. Compute $r = (g^k \bmod p) \bmod q$.
4. Compute $s = (h(m) + rx)k^{-1} \bmod q$, where $kk^{-1} \equiv 1 \pmod q$ and $h$ denotes the
   hash function SHA-1[12].
5. If $r = 0$ or $s = 0$ then go to 2.
6. Compute $u = h(m)s^{-1} \bmod q$, where $ss^{-1} \equiv 1 \pmod q$.
7. Compute $v = rs^{-1} \bmod q$.
8. Compute $w = (g^u y^v \bmod p) \bmod q$.
9. If $w = r$ then return ($r$, $s$).
10. $i \leftarrow i + 1$
11. If $i > B$ then return FAILURE.
12. Go to 2.
```

□

As we can see, the algorithm describes formally what a programmer would do naturally if she was asked to implicitly verify every signature made whether it is valid or not before letting it go out from a cryptographic module. Steps 2 to 4 cover the signature generation, while steps 5 to 9 do the signature verification. Both parts are written according to [8]. Another thing that would the programmer do naturally in such a situation is to employ an automatic repeat function which would retry the signing operation several times before the algorithm echoes a failure to a calling process. This constitutes the repeat part of TAR which is driven by the boundary denoted as B. Note that for a small value of the boundary (circa $B \leq 20$), such an algorithm can also originate due to a user activity: The user, for instance, wants to send a signed e-mail, while the device says that there is something wrong about a signing module. We can reasonably expect that she would try to sign her message several times before she gives it up. The more eager the user is the higher B we get.

2.2. Embedding the Fault Side Channel

On a first glimpse, Algorithm 1 described above can be regarded as being resistant against fault attacks, since no faulty signature can leave perhaps the innermost place of the cryptographic module. Such a reasoning which could be inspired by typical symptoms of fault attacks on RSA (c.f. [1], [2], [13], [14]), can, however, be terribly misleading here. An example of fault attack that passes undetected in such a situation can be found in [11]. A fault attack that can be even right allowed thanks to relying on such a "fault tolerant" algorithm was then presented in [4]. A brief description of the attack follows.

Let d be an integer, such that $d \mid p - 1$ and $\gcd(d, q) = 1$. Furthermore, let β be an integer, $1 < \beta < p$, of order $\mathrm{ord}(\beta) = d$ in $\boldsymbol{Z}_p^*$. Now, let us suppose that an attacker substitutes the value of $g' = g\beta \bmod p$ in place of g in the private key record in Algorithm 1. Such a change can be theoretically possible, since g is a part of non-secret public parameters whose protection architects often tend to underestimate. For instance, in the CryptoAPI subsystem of the MS Windows platform, there is a function CryptSetKeyParam with the parameter KP_G reserved for such a purpose [15]. It is left up to designers of cryptographic modules how to implement this function and whether to allow such modifications at all. There is, however, no warning about how dangerous this functionality can be. Therefore, we may reasonably assume that at least some architects will allow the attacker to freely change the value of g. Several problems with integrity of a key material were also identified by Clulow for the PKCS#11 security standard platform [13]. There was also a successful attack based on DSA public parameters modification described by Klíma and Rosa in [14]. We shall, therefore, fully anticipate the possibility of such a modification when we discuss security aspects of a particular signing procedure.

Now, let us denote r' and s' the variables from Algorithm 1 computed for a substituted value of $g' = g\beta \bmod p$. We can write:

$$r' = (g^k\beta^k \bmod p) \bmod q, \quad (3)$$

$$s' = (h(m) + r'x)k^{-1} \bmod q,\ kk^{-1} \equiv 1 \pmod q). \quad (4)$$

Let us assume that in step 8, the module uses the value of the correct generator g. That means that the attacker will not affect the public key record which is usually loaded from an independent storage – possibly from a user's public key certificate [3]. Note that the attack is possible even if the attacker changes the generator in both of the public and private key records [4]. Let us denote u' and v' the values computed in steps 6 and 7, respectively. Using their definitions together with Eq. (4) over GF(q), it follows that:

$$u' + v'x \equiv h(m)(s')^{-1} + r'(s')^{-1}x \equiv (h(m) + r'x)(s')^{-1} \equiv k \pmod q. \quad (5)$$

Now, let us denote w' the value computed in step 8. Since the algorithm uses the unaffected value of g of order q, we can use Eq. (5) and write:

$$w' = (g^{u'}y^{v'} \bmod p) \bmod q = (g^{u'+v'x} \bmod p) \bmod q = (g^k \bmod p) \bmod q. \quad (6)$$

Basing on Eqs. (3) and (6), we can rewrite the condition the signature (r', s') must pass in step 9 as:

$$w' = r' \Leftrightarrow (g^k \bmod p) \bmod q = (g^k\beta^k \bmod p) \bmod q. \quad (7)$$

Since we can neglect an influence of "inner" collisions in the mapping $\varphi(k) = (g^k\beta^k \bmod p) \bmod q$ (c.f. [16]), we can claim that with a probability close to 1 the following condition is necessary and sufficient to release the signature (r', s') in step 9:

$$\beta^k \bmod p = 1, \text{ i.e. } k \equiv 0 \pmod d). \quad (8)$$

We see that the attacker gets nontrivial direct information about the nonce *k* whenever Algorithm 1 releases a signature pair (*r*', *s*'), since she knows that whatever the nonce is, it must be an integer divisible by a known value *d*. This creates a vital side channel that she can use to recover the whole value of the private key using a slightly modified approach from [9]. A detailed description of the computation is given in [4]. Here, we present the following table showing certain experimental results.

Table 1. Experimental fault attacks on several randomly chosen DSA instances.

Exp. No.	**Divisor *d***	**#Signatures**	**#Signatures Total**	**Exp. Duration**
1	12	70	880	182 s
2	12	55	688	66 s
3	15	61	923	120 s
4	12	55	649	63 s
5	2	weak channel	N/A	N/A
6	14	48	550	44 s
7	22	46	912	67 s
8	12	55	832	76 s
9	2	weak channel	N/A	N/A
10	12	65	621	118 s

Each numbered row of Table 1 corresponds to an experimental fault attack on a particular randomly generated DSA instance. The divisor *d* was chosen automatically by the attacking program to be small enough while producing a usable side channel for the attack. Small values are desirable, since the probability that step 9 releases a particular signature can be estimated as d^{-1}. The number of TAR iterations follows a geometric distribution, so the expected value and the variance of the number of signatures computed before releasing a valid signature is then $EX = d$ and $\mathrm{Var}(X) = d(d - 1)$, respectively. Sometimes, there was no suitable divisor found with respect to a realistically tight boundary of repetitions (several hundreds). The number of signatures in Table 1 denotes the number of valid signatures used for a successful private key reconstruction. The total number of signatures illustrates the number of invalid signatures produced and discarded within TAR. The duration of each experiment shows how effective the whole attack is, since this time covers the DSA instance generation, the attack preparation, the signatures generation, and the private key reconstruction. The platform used for the experiments was a general office notebook with Pentium M/1.5 GHz and Windows 2000. The code was written in C++ and supported by the Shoup's NTL library [17].

3. Conclusion

We saw that the "Test and Repeat" paradigm cannot be regarded as a robust countermeasure against fault attacks, since there are realistic strategies that pass undetected by it. We also saw that there are attacks which can be even right allowed thanks to relying on the "power" of this approach. Therefore, despite being a bit paradoxical on a first glimpse, we shall use it very carefully in a cryptographic modules design. Of course, this is not to say that we shall not use it at all. We just shall bear on our minds that we must not rely solely on this approach and that we have to design and implement it properly. The caution mainly addresses the phase of a design verification in which we shall check every possible attack scenario to see whether our implementation can resist it or not. Otherwise, the situation about overall cryptanalytical attacks can become even worse, since some of them may become hidden and accelerated, some of them even right allowed.

Acknowledgements

I am grateful to Vlastimil Klíma for many helpful comments and inspiring suggestions on the previous versions of the paper.

References

[1] Boneh, D., DeMillo, R.-A., and Lipton, R.-J.: *On the Importance of Checking Cryptographic Protocols for Faults*, in Proc. of EUROCRYPT '97, pp. 37-51, Springer-Verlag, 1997.

[2] Rosa, T.: *Modern Cryptology – Standards Are Not Enough*, Ph.D. Thesis, 2004.

[3] Menezes, A.-J., van Oorschot, P.-C., and Vanstone, S.-A.: *Handbook of Applied Cryptography*, CRC Press, 1996.

[4] Rosa, T.: *Lattice-based Fault Attacks on DSA - Another Possible Strategy*, in Proc. of Security and Protection of Information 2005, pp. 91-96, Brno, 2005.

[5] Simons, G.-J.: *The History of Subliminal Channels*, IEEE Journal of Selected Areas in Communications, Vol. 16, No. 4, pp. 452-462, April 1998.

[6] Bishop, M.: *Computer Security – Art and Science*, Addison-Wesley, 2003.

[7] FIPS PUB 140-2: *Security Requirements for Cryptographic Modules*, National Institute of Standards and Technology, May 25 2001.

[8] FIPS PUB 186-2: *Digital Signature Standard (DSS)*, National Institute of Standards and Technology, January 27 2000, updated: October 5 2001.

[9] Nguyen, P.-Q., and Shparlinski, I.-E.: *The Insecurity of the Digital Signature Algorithm with Partially Known Nonces*, Journal of Cryptology, Vol. 15, No. 3, pp. 151-176, Springer-Verlag, 2002.

[10] Boneh, D., and Venkatesan, R.: *Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes*, in Proc. of CRYPTO '96, pp. 129-142, Springer-Verlag, 1996.

[11] Naccache, D., Nguyen, P.-Q., Tunstall, M., and Whelan, C.: *Experimenting with Faults, Lattices and the DSA*, in Proc. of Public Key Cryptography – PKC'05, pp. 16-28, Springer-Verlag, 2005.

[12] FIPS PUB 180-1: *Secure Hash Standard (SHA-1)*, National Institute of Standards and Technology, January 2001.

[13] Clulow, J.: *On the Security of PKCS #11*, in Proc. of CHES 2003, pp. 411-425, Springer-Verlag, 2003.

[14] Klíma, V., and Rosa, T.: *Attack on Private Signature Keys of the OpenPGP format, PGP$^{(TM)}$ Programs and Other Applications Compatible with OpenPGP*, IACR ePrint archive, 2002/076, http://eprint.iacr.org, 2001.

[15] Microsoft CryptoAPI, *MSDN Library*, October 2001.

[16] Brown, D.-R.-L.: *Generic Groups, Collision Resistance, and ECDSA*, IEEE 1363 report, February 2002.

[17] Shoup, V.: *Number Theory C++ Library (NTL)*, http://www.shoup.net/ntl/.