Improving Divide and Conquer Attacks Against Cryptosystems by Better Error Detection / Correction Strategies

Werner Schindler[1], François Koeune[2] and Jean-Jacques Quisquater[2]

[1] Bundesamt für Sicherheit in der Informationstechnik (BSI)
Godesberger Allee 185–189
53175 Bonn, Germany
Werner.Schindler@bsi.bund.de
[2] Université catholique de Louvain
Place du Levant 3
1348 Louvain-la-Neuve, Belgium
{fkoeune,jjq}@dice.ucl.ac.be

Abstract. Divide and conquer attacks try to recover small portions of cryptographic keys one by one. Usually, a wrong guess makes subsequent ones useless. Hence possible errors should be detected and corrected as soon as possible. In this paper we introduce a new (generic) error detection and correction strategy. Its efficiency is demonstrated at various examples, namely at a power attack, two timing attacks against RSA implementations with and without Chinese Remainder Theorem, and a timing attack against the future AES (Rijndael). As the design of efficient countermeasures requires a good understanding of an attack's actual power, the possible improvement induced by sophisticated error detection and correction should not be neglected. Although divide and conquer attacks are typical for side-channel attacks, we would like to stress that they are not restricted to that field, as will be illustrated by Siegenthaler's attack.

Keywords: Error detection, error correction, timing attack, power attack.

1 Introduction

Cryptographic algorithms (encryption schemes, random generators, ...) often gather their security on one (or a few) secret parameter(s), whereas the rest of the design is left public.

To seize this secret parameter (usually denoted as the *key*), a frequent attack scenario assumes that the pirate is able to observe the output of the algorithm, possibly with access – or even control – of this algorithm's input during a limited period of time. The attacker will use these observations to deduce information on the key.

In particular, divide and conquer attacks basically consist in dividing the key into smaller pieces, whose size makes exhaustive search possible, and then

handling these pieces separately. Such attacks are very efficient, but will of course only be feasible provided that it is possible to guess parts of the key separately. In other words, it must be possible, with reasonable probability, to confirm or invalidate a partial key guess without knowing the other parts of the key.

In many cases, a wrong partial key guess makes subsequent ones worthless. Consequently, it is desirable to be able to validate the guesses made so far. This suggests the following iterated process:

1. guess a part of the key
2. check whether guess is correct so far (error detection);
3. if yes, repeat step 1 with next part of the key;
4. otherwise, identify probable error position(s) among previously guessed parts (error location), and go back to step 1 trying another guess for that part (error correction).

Many different attacks can be put in the divide and conquer class, and the relative importance of the different parts (guess, error detection, ...) may vary greatly from one to another. However, these parts are usually more or less independent, and can therefore be subject to independent improvements, any of which would result in a global performance improvement.

This paper focuses on the error management [1] part. Error management faces several efficiency constraints. First, it must keep sample size small: a simple error check, for example, consists in repeating Step 1 with a new observation; however, this reduces the efficiency of the attack by at least a factor 2, which is not acceptable for many realistic scenarios. Second, it must be time-efficient, in the sense that a wrong guess must be identified as quickly as possible; but this must be counterbalanced with the risk of hindering the attack's success: a too restrictive strategy involves the risk of definitively rejecting a guess that was correct, with the consequence that the attack will fail completely, whereas a too permissive strategy will make the attack longer, possibly up to impractical running time.

In this paper we do not concentrate on the attacks themselves, i.e. on strategies to guess parts of the key (Step 1) and on countermeasures which prevent these attacks but on error detection and error correction strategies (Steps 2 and 4). In particular, we mainly suggest a new type of error detection strategy which may be characterized as a *three-option decision strategy.* Everytime the error detection is applied, it can either:

2.a. Validate the correctness of previous partial key guesses up to a certain point (which need not necessarily be the current one). This yields a "confirmed index" (i.e. we definitively assume that that part of the key is correct) which will facilitate later error locations/corrections, as only the partial key guesses which were derived *after* the actual confirmed index will be later considered.

[1] The idea that errors could be identified and corrected in a timing attack was first mentioned by Kocher [7].

2.b. Conclude an error has occurred between this point and the last confirmed index. In this case, we enter the error-correction phase.
2.c. If there are no convincing indicators for any of these two cases, do not conclude. In this case, we simply continue the attack, and postpone decision to a future error detection step.

This concept is rather generic and can be adapted to various types of attacks. We will consider five examples which may seem to be very different at first sight.

Section 2 briefly introduces the subject by presenting an extreme example of divide and conquer attack, in which the error correction and detection strategy is so efficient that it constitutes the most important part of the attack. Section 3, which is the main part of this paper, then develops the error management strategy in detail, on a timing attack example. In this example, the sophisticated error detection and correction strategy, together with an optimized bit estimation strategy, allowed to improve the efficiency of a timing attack on RSA ([6]) by about factor 50. Section 4 shows how a similar technique can be applied to a timing attack against another RSA implementation (using the Chinese Remainder Theorem (CRT) and thus being resistant against the attacks considered in [6] and [13]). Section 5 treats of a timing attack on a careless AES (Rijndael) implementation (cf. [8]) which is atypical as the key parts principally can be guessed independently and errors do not influence the later guesses. However, also in this case an efficient error location strategy is evaluated. We would like to insist on the fact that the proposed error management strategy is in fact much more general, and can be applied to many other divide and conquer attacks. Although the examples developed here correspond to physical attacks (timing attack, power analysis), divide and conquer attacks may also appear in algorithmically based attacks. This will be sketched in Sect. 6.

We point out that this paper was not written as a "manual" for potential attackers but shall help designers to assess the realism and the true threat of certain divide and conquer attacks.

2 Messerges et al.'s power analysis

Messerges et al.'s Multiple Exponent, Single Data (MESD) attack ([10]) corresponds to some extreme form of divide and conquer attack. Although this attack's efficiency leaves very little room for improvement through a new error-detection policy, we believe that, due to its simplicity, it constitutes a good example to start with. The reader may consider it as a witness of how much an adequate error-detection policy can improve an attack's performances.

The context is that of a smart card accepting to exponentiate a constant value with user-supplied exponents[2]. The attacker is able to measure the power consumption of that card, and tries to guess the secret exponent used by that

[2] We will not discuss this attack's realism here; see [10] for a brief discussion on the subject.

card. The attack could be described as follows. Assume the attacker already knows the most significant t bits $k_{w-1}, \ldots, k_{w-t}$ of the key:

guessing phase attacker builds an exponent $e = k_{w-1} \cdots k_{w-t} r_{w-t-1} \cdots r_0$, where bits r_i are chosen at random, and submits this exponent to the card;

error detection/location attacker compares the power consumption of exponentiation with e to the power consumption of exponentiation with the secret key k, and notes the position s of the first bit after which the two curves differ (due to measurement errors, this may require averaging on several exponentiations with same input);

error correction attacker builds a new exponent $e' = k_{w-1} \cdots k_s \overline{k}_{s-1} r'_{s-2} \cdots r'_0$, where $\overline{k}_i$ denotes $1 - k_i$ and the r'_i are chosen at random.

In this case, the error location method is so efficient, allowing to point the error position almost exactly, that the guessing process can be limited to its simplest form (random guess). Similarly, this precision in error location allows a very simple error correction (bit inversion).

3 Timing attack against RSA without CRT

In this section we describe a timing attack against RSA signature. The attack ([6]) was initially developed against a preliminary version of the Cascade ([2]) smart card, although it would work equally well against many other modular exponentiation algorithms without CRT (see [6, 13]). Our approach – the optimal decision strategy derived in [13] combined with a very efficient error detection and correction strategy – increases the efficiency of the attack by about factor 50. As this attack constitutes a systematic approach to exploit side-channel information in an optimal way, we will describe the attack and the development of our error management policy in detail.

Remark 1. (i) The attack presented here is a pure timing attack, in the sense that the only information we dispose of is a set of messages and, for each of them, the total time required for signature;
(ii) in view of [6], the final version of the Cascade cryptographic library was later modified to resist against timing attacks ([5]).

3.1 Definitions and Mathematical Background

To compute $y^d (\bmod M)$ the Cascade chip uses the simple square and multiply algorithm. Modular multiplications are carried out with Montgomery's algorithm ([11]). In its simplest variant $R := 2^\omega > M$ where ω fits to the device's hardware architecture. Let $R^{-1} \in Z_M := \{0, \ldots, M-1\}$ denote the multiplicative inverse of R in Z_M, i.e. $RR^{-1} \equiv 1 \pmod M$. The integer $M' \in Z_R$ satisfies the integer equation $RR^{-1} - MM' = 1$. To simplify our notation we introduce the functions $\Psi, \Psi_*\colon Z \to Z_M$ defined by $\Psi(x) := xR (\bmod M)$ and

$\Psi_*(x) := xR^{-1}(\bmod M)$. For $a' = \Psi(a)$ and $b' = \Psi(b)$ Montgomery's algorithm returns $s := \Psi_*(\Psi(a)\Psi(b)) = \Psi(ab)$.

Montgomery's algorithm

$z := a'b'$;
$r := (z(\bmod R))M' \pmod R$
$s := \frac{z+rM}{R}$
if $(s \geq M)$ then $s := s - M$
return $s \quad (= \Psi_*(a'b') = a'b'R^{-1})$

Algorithm 1 (square and multiply using Montgomery's algorithm)

```
temp := Ψ(y);
for i=w-2 downto 0 do {
    temp := Ψ*(temp^2);
    if (d_i = 1)  temp := Ψ*(temp * Ψ(y));
}
return Ψ*(temp);
```

The secret exponent d has binary representation $(d_{w-1}d_{w-2}\ldots d_0)_2$ where d_{w-1} denotes its most significant bit. Further, $\mathrm{h}am(d)$ denotes the Hamming weight of d. The subtraction $s := s - M$ in Montgomery's algorithm is called *extra reduction* while $y \mapsto \Psi(y)$ and temp $\mapsto \Psi_*(\text{temp})$ are the *pre-multiplication* and the *post-multiplication*. For any $a', b' \in Z_M$ we have $\mathrm{T}ime(\Psi_*(a', b')) = c$ if no extra reduction is necessary and $= c + \mathrm{c_{ER}}$ else. The sources of our timing attack are time differences which are caused by different numbers of extra reductions within the for-loop of Algorithm 1.

Remark 2. Many implementations (among which Cascade) use a more efficient multiprecision variant of Montgomery's algorithm (see e.g. [9], Algorithm 14.36) than the one listed above. This influences the absolute value of the constants c and $\mathrm{c_{ER}}$ but *not* the fact whether an extra reduction is necessary ([14], Remark 1). We hence clearly analyze the simplest variant of Montgomery's algorithm described above.

Let $t := \mathrm{T}ime(y^d(\bmod M))$. For a sample $y_{(1)}, \ldots, y_{(N)} \in Z_M$ the attacker measures $\widetilde{t}_{(1)} := t_{(1)} + t_{\mathrm{Err}(1)}, \ldots, \widetilde{t}_{(N)} := t_{(N)} + t_{\mathrm{Err}(N)}$ where $t_{\mathrm{Err}(j)}$ denotes the measurement error. More precisely,

$$\widetilde{t}_{(j)} = t_{\mathrm{Err}(j)} + t_{S(j)} + (w + \mathrm{h}am(d) - 2)c + r_{(j)}\,\mathrm{c_{ER}} \tag{1}$$

where $t_{S(j)}$ denotes the time needed for set-up operations such as input, output, increasing the loop variable, evaluating the if-statements and, above all, the pre- and post-multiplication. Finally, $r_{(j)}$ denotes the number of extra reductions needed within the for-loop of Algorithm 1, i.e. $r_{(j)} = w_{1(j)} + w_{2(j)} + \cdots$ where $w_{i(j)} = 1$ if the ith Montgomery multiplication in Algorithm 1 requires an extra reduction for basis $y_{(j)}$ while $w_{i(j)} = 0$ else. From (1) we derive the "discretized running time"

$$\widetilde{t}_{d(j)} := \frac{\widetilde{t}_{(j)} - t_{S(j)} - (w + \mathrm{h}am(d) - 2)c}{\mathrm{c_{ER}}} \; . \tag{2}$$

If $t_{\mathrm{Err}(j)} = 0$ (i.e. for exact time measurement), $\widetilde{t}_{d(j)}$ equals $r_{(j)}$, the total number of extra reductions needed within the for-loop of Algorithm 1. The values $w_{1(j)}, w_{2(j)}, \ldots$ can be interpreted as realizations (i.e. values assumed by) of

a particular non-stationary sequence of random variables $W_{1(j)}, W_{2(j)}, \ldots$ which are closely related with the Montgomery multiplications within Algorithm 1 (cf. [13], Sect. 6). (Numerous empirical experiments confirmed perfectly the suitability of this mathematical model.) In particular, the definition of $W_{i(j)}$ explicitly depends on the basis $y_{(j)}$ and whether the ith Montgomery multiplication within the for-loop in Algorithm 1 is a squaring (shortly: $type(i) =' Q'$) or a multiplication with $\Psi(y_{(j)})$ (shortly: $type(i) =' M'$), resp. To derive an optimal decision strategy, the sequence $W_{1(j)}, W_{2(j)}, \ldots$ has to be studied first. We briefly give the main results. For a proof the interested reader is referred to [13] (Lemma 6.3). In particular, the expectation of $W_{i(j)}$ (which equals the probability for an extra reduction in the ith Montgomery multiplication for basis $y_{(j)}$) is given by

$$\mathrm{E}(W_{i(j)}) = \begin{cases} p_* := \frac{1}{3}\frac{M}{R} & \text{if } type(i) =' Q' \\ p_j := \frac{\Psi(y_{(j)})}{2M}\frac{M}{R} & \text{if } type(i) =' M'. \end{cases} \tag{3}$$

The covariance $\mathrm{Cov}(W_{i(j)}, W_{i+1(j)})$ equals

$$\begin{cases} \mathrm{cov}_{\mathrm{MQ(j)}} := 2p_j^3 p_* - p_j p_* & \text{if } (type(i), type(i+1) = ('M','Q') \\ \mathrm{cov}_{\mathrm{QM(j)}} := \frac{9}{5} p_j p_*^2 - p_j p_* & \text{if } (type(i), type(i+1) = ('Q','M') \\ \mathrm{cov}_{\mathrm{QQ}} := \frac{27}{7} p_*^4 - p_*^2 & \text{if } (type(i), type(i+1) = ('Q','Q') \end{cases} \tag{4}$$

whereas $W_{i(j)}$ and $W_{h(j)}$ are independent if $|i-h| > 1$. (5)

3.2 Guessing d_k

Our attack estimates the exponent bits $d_{w-1}, \ldots, d_0$ successively. We assume that the attacker has already estimated the exponent bits $d_{w-1}, \ldots, d_{k+1}$ and that his estimators $\widetilde{d}_{w-1}, \ldots, \widetilde{d}_{k+1}$ have been correct. From these estimators he determines the respective temp values before the if-statement in the for-loop for $i = k$ for all bases $y_{(1)}, \ldots, y_{(N)}$ in his sample. Finishing the exponentiation ${y_{(j)}}^d (\mathrm{mod}\, M)$ still requires k squarings. The number m of remaining multiplications with $\Psi(y_{(j)})$ results from w, $ham(d)$ and $\widetilde{d}_{w-1}, \ldots, \widetilde{d}_{k+1}$. Subtracting the extra reductions already carried out from the discretized running time $t_{d(j)}$ yields the *remaining discretized running time* $\widetilde{t}_{\mathrm{drem}(j)}$ which is interpreted as realization of random variable

$$\widetilde{T}_{\mathrm{drem}(j)} = T_{\mathrm{dErr}(j)} + W_{w+ham(d)-k-m-1(j)} + \cdots + W_{w+ham(d)-2(j)}. \tag{6}$$

It is reasonable to assume that the (random) measurement error is independent of the running time and hence that $T_{\mathrm{dErr}(j)}$ is independent of $W_{w+ham(d)-k-m-1(j)} + \cdots + W_{w+ham(d)-2(j)}$. Further, we assume that it is normally distributed with expectation 0 and variance $\alpha^2 := \sigma_{\mathrm{Err}}^2 / {\mathrm{c_{ER}}}^2$.

From the 4-tuples $(\widetilde{t}_{\mathrm{drem}(1)}, u_{M(1)}, u_{Q(1)}, t_{Q(1)}), \ldots, (\widetilde{t}_{\mathrm{drem}(N)}, u_{M(N)}, u_{Q(N)}, t_{Q(N)})$ the attacker derives an estimator $\widetilde{d}_k$ for the unknown exponent bit d_k. Here $t_{Q(j)}$ (resp., $u_{M(j)}$, resp. $u_{Q(j)}) \in \{0,1\}$ equals 1 iff the next Montgomery multiplication (i.e., squaring, resp. multiplication by $\Psi(y_{(j)})$, resp. squaring after this multiplication) requires an extra reduction. Formally, this can be interpreted

a statistical decision problem where the attacker has to decide between the two hypotheses $\theta = 0$ (corresponding to the case $d_k = 0$) and $\theta = 1$ (corresponding to $d_k = 1$) (cf. [13]). Theorem 1 gives the optimal decision strategy for the next bit value[3], i.e. a strategy which minimizes the probability that $\widetilde{d}_k \neq d_k$. As before, the letter N stands for the sample size.

Notations. Within theorem 1 we use the abbreviations
$hn(0,j) := (k-1)p_*(1-p_*)+mp_j(1-p_j)+2(m-1)\mathrm{cov}_{\mathrm{MQ(j)}}+2(m-1)\mathrm{cov}_{\mathrm{QM(j)}}+ 2(k-m-1)\mathrm{cov}_{\mathrm{QQ}} + 2\frac{k-m}{k-1}\mathrm{cov}_{\mathrm{QM(j)}} + 2\frac{m-1}{k-1}\mathrm{cov}_{\mathrm{QQ}} + \alpha^2$,
$hn(1,j) := (k-1)p_*(1-p_*) + (m-1)p_j(1-p_j) + 2(m-2)\mathrm{cov}_{\mathrm{MQ(j)}}+ 2(m-2)\mathrm{cov}_{\mathrm{QM(j)}}+2(k-m)\mathrm{cov}_{\mathrm{QQ}}+2\frac{k-m+1}{k-1}\mathrm{cov}_{\mathrm{QM(j)}}+2\frac{m-2}{k-1}\mathrm{cov}_{\mathrm{QQ}} +\alpha^2$,
$ew(0,j \mid b) := (k-1)p_* + mp_j + \frac{k-m}{k-1}(p_{*Q(b)} - p_*) + \frac{m-1}{k-1}(p_{jQ(b)} - p_j)$
$ew(1,j \mid b) := (k-1)p_* + (m-1)p_j + \frac{k-m+1}{k-1}(p_{*Q(b)} - p_*) + \frac{m-2}{k-1}(p_{jQ(b)} - p_j)$
with
$p_{*Q(1)} := \frac{27}{7}p_*^3$, $p_{*Q(0)} := \frac{p_*-p_*p_{*Q(1)}}{1-p_*}$, $p_{jQ(1)} := \frac{9}{5}p_*p_j$ and $p_{jQ(0)} := \frac{p_j-p_*p_{jQ(1)}}{1-p_*}$.

Theorem 1. *(i) Assume that the estimators $\widetilde{d}_{w-1}, \ldots, \widetilde{d}_{k+1}$ have been correct and that $d_k + \cdots + d_0 = m$, i.e. for each exponentiation m Montgomery multiplications of type 'M' still have to be carried out. Let*

$$\psi_{N,d} : (\mathbb{R}\times\{0,1\}^3)^N \to \mathbb{R}, \qquad \psi_{N,d}((\widetilde{t}_{\mathrm{drem}(1)}, u_{M(1)}, \ldots, u_{Q(N)}, t_{Q(N)})) := -\frac{1}{2}\sum_{j=1}^{N}\left(\frac{\left(\widetilde{t}_{\mathrm{drem}(j)} - t_{Q(j)} - ew(0,j \mid t_{Q(j)})\right)^2}{hn(0,j)} - \frac{\left(\widetilde{t}_{\mathrm{drem}(j)} - u_{M(j)} - u_{Q(j)} - ew(1,j \mid u_{Q(j)})\right)^2}{hn(1,j)}\right).$$

Then the deterministic decision strategy $\tau_d \colon (\mathbb{R}\times\{0,1\}^3)^N \to \{0,1\}$ defined by

$$\tau_d = 1_{\psi_{N,d}<\log(\frac{m-1}{k-m+1})+\frac{1}{2}\sum_{j=1}^{N}\log(1+c_j)} \qquad \text{with} \quad c_j := \frac{hn(0,j)-hn(1,j)}{hn(1,j)} \tag{7}$$

is optimal.

A nice property of the optimal decision strategy described above is that it allows to detect errors. It can be shown that, after an error has occurred (i.e. bit $\widetilde{d}_{k^*} \neq d_{k^*}$), the probability $\mathrm{Prob}(\widetilde{\mathrm{d}}_{\mathrm{k}} = 1)$ to guess 1 for subsequent bits is about 0.20 (although the exact value is parameter-dependent and thus changes during the attack). The proof of this fact, as well as precise probabilities, can be found in [13] (Theorem 6.5); for the sake of simplicity, however, we will skip these – rather complex – expressions here, and focus on the way we can exploit this error witness.

[3] For a proof of this theorem, the interested reader is referred to [13] (proof of Theorem 6.5(i)).

Remark 3. It can also be shown that the error probability $\mathrm{Prob}(\widetilde{\mathrm{d}}_\mathrm{k} \neq \mathrm{d}_\mathrm{k})$ decreases as the attack proceeds. To quantify this probability, the distribution of $Z := \psi_{N,d}(\widetilde{T}_{\mathrm{drem}(1)}, U_{M(1)}, \ldots, U_{Q(N)}, T_{Q(N)})$ has to be determined for both alternatives $\theta = 0$ and $\theta = 1$. (We interpret $\widetilde{t}_{\mathrm{drem}(1)}, \ldots, t_{Q(N)}$ as realizations of specific random variables $\widetilde{T}_{\mathrm{drem}(1)}, \ldots, T_{Q(N)}$.) The minimal sample size which guarantees a particular error probability (e.g., 0.01) is essentially linear in k. Once again, we refer the interested reader to [13] (Theorem 6.5) or to [15] (theorem without proof) for precise expressions.

Let us illustrate these two facts (low probability to guess a 1 after an error has occurred and decreasing error probability as the attack proceeds), by an example (cf. [13]):

Example 1. In (a) and (b) below we assume that the estimators $\widetilde{d}_{w-1}, \ldots, \widetilde{d}_{k+1}$ have been correct. For randomly chosen bases $y_{(1)}, \ldots, y_{(N)}$ the following inequalities hold in good approximation.
Let $M/R = 0.7$, $\alpha^2 = 0$, $N \geq 5620$, and further ....
(a) ... $(k, m) = (510, 255)$. Then $\mathrm{Prob}(\widetilde{\mathrm{d}}_\mathrm{k} \neq \mathrm{d}_\mathrm{k}) \leq 0.01$.
(b) ... $(k, m) = (440, 220)$. Then $\mathrm{Prob}(\widetilde{\mathrm{d}}_\mathrm{k} \neq \mathrm{d}_\mathrm{k}) \leq 0.0064$.
(c) ... $(k^*, m^*, k, m) = (505, 250, 470, 235)$.
Then $\mathrm{Prob}(\widetilde{\mathrm{d}}_\mathrm{k} = 1) \leq 0.1879$ *and* $\mathrm{Prob}(\psi_{\mathrm{N,d}} < \mathrm{E}_{\theta=1}(\mathrm{Z}) \mid \widetilde{\mathrm{d}}_\mathrm{k} = 1) \leq 0.0170$.

The error detection and correction strategy described below is more efficient than its pre-variant described in [13]. Roughly speaking, to estimate the exponent bits the attacker derives a sequence of $\psi_{N,d}$-values on which his decisions are based on. These values themselves can be interpreted as realizations of random variables (cf. Fig. 1 below) whose distribution changes noticeably after the first wrong guessing.

3.3 Error Detection

The following diagram illustrates the facts we will base our error detection on. The curves g_0, g_1 and g_f are the density functions of $Z := \psi_{N,d}(\cdot)$ defined in Theorem 1 if $d_k = 0$, $d_k = 1$, or if $\widetilde{d}_{k'}$ was false for any $k' > k$, resp. (In particular, g_0, g_1 and g_f are normal densities; cf. the proof of Theorem 6.5 in [13].) As we have seen, in the latter case, it is unlikely to derive $\widetilde{d}_k = 1$. If this yet happens the $\psi_{N,d}(\cdot)$-value is $\geq E_{\theta=1}(Z)$ (the expectation of Z if $\theta = 1$) with high probability. Both observations will be the basis of our error detection and correction strategy. The arrow in Fig. 1 points to the area corresponding to the probability that $\psi_{N,d} < E_{\theta=1}(Z)$.

The task of an error detection strategy is to check whether an error has occurred. The error probability decreases when k decreases so that estimation errors usually occur at the beginning of the attack, at least if $\alpha^2 = 0$ and if the attacker knows the values $w, \mathrm{h}am(d), c, \mathrm{c}_{\mathrm{ER}}$ and t_S or has guessed them

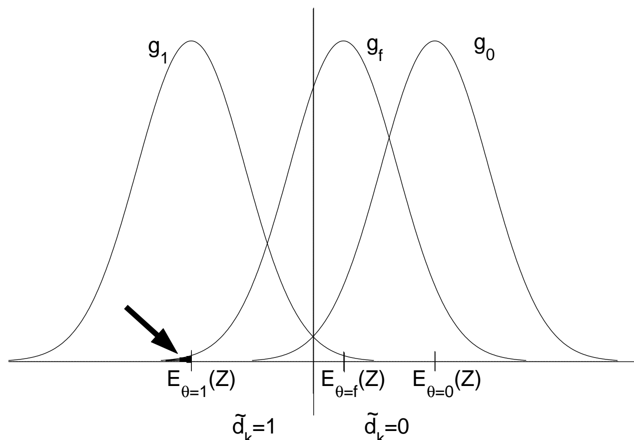

Fig. 1. Density functions of $\psi_{N,d}$ depending on d_k

correctly (cf. Subsect. 3.5). Note that there is also a possibility for errors to occur at the very end of the attack if the parameter guesses are not absolutely correct; this, however, is no serious harm as the few final exponent bits may be checked exhaustively: we will thus leave this case aside.

The basic idea of our error-detection strategy is the following: before guessing a new bit d_k, we will look at a "window" of the f preceding bits, and test the two (non-complementary!) hypotheses:
(A) The estimators $\widetilde{d}_{w-2}, \ldots, \widetilde{d}_{k+1}$ are correct.
(B) There is an index $k' > k + f$ for which $\widetilde{d}_{k'}$ is wrong.

Roughly speaking, a false hypothesis (A) will be witnessed by an unusually large number of 0s in the window, whereas a large number of 1s tends to infirm hypothesis (B). If we can neither reject (A) nor (B), then we simply increase the window's length and test the hypotheses again; if the window reaches its maximum size with yet no decision being possible, the attack advances one position, deciding the next bit d_k as usual.

If hypothesis (B) is rejected, then we can set up a *confirmed index* at position $k + f + 1$, meaning that no error has occurred before this point. Therefore we will not try to modify any bit located before it, i.e. with greater or equal index than a confirmed index.

If hypothesis (A) is rejected ("alarm"), we must choose one index $k'' \in \{\text{con} - 1, \ldots, k + 1\}$, revert that decision, and start the attack from that point over again. The attack will continue until either:

- a new confirmed index is established, in which case the algorithm "forgets" an alarm had occurred and continues the attack (having corrected the error at k''), or

– a further alarm occurs, in which case we conclude the index k'' was not the first error position; we then choose a new index in the same set $\{\mathrm{con}-1,\ldots,k+1\}$ and restart from that point.

To determine the threshold values that will conduct to the rejection of hypotheses (A) and (B), we interpret the sum $nones := \widetilde{d}_{k+f} + \cdots + \widetilde{d}_{k+1}$ as a realization of a binomial random variable X_1. If (A) is true, then this variable's distribution is $X_1 \sim B(f, 0.5)$; we therefore reject (A) if $\mathrm{Prob(X_1 \leq nones)} < \mathrm{p_{al}} := 0.0001$.

The condition to reject (B) is a bit more complicated: in a setup-step, we used Theorem 6.5(iii) ([13]) to obtain rough approximators p_{err} and p_{len}, resp., for the average probabilities $\mathrm{Prob}(\widetilde{\mathrm{d}}_{\mathrm{k}} = 1)$ or $\mathrm{Prob}(\psi_{\mathrm{N,d}} < \mathrm{E}_{\theta=1}(\mathrm{Z}) \mid \widetilde{\mathrm{d}}_{\mathrm{k}} = 1)$, resp., within the initial stage of the attack under the condition that $\widetilde{d}_{k'}$ was false for a $k' > k$. We reject (B) if $len := |\{k+1 \leq i \leq k+f \mid \widetilde{d}_i = 1, \psi_{N,d} < E_{\theta=1}(Z)\}| \geq 2$ and $\mathrm{Prob(X_2 \geq len)} < \mathrm{p_{con}} := 0.0005$ where X_2 is $B(nones, p_{\mathrm{len}})$-distributed. The choice of the probabilities p_{al} and p_{con} was somewhat arbitrary. The values 0.0001 and 0.0005 turned out to be suitable.

3.4 Error Correction

We have not detailed yet in which order the successive k'' are chosen during the successive alarms: after a "new" alarm (i.e. an alarm when any preceding one has been "forgotten") the indices $\{\mathrm{con}-1,\ldots,k+1\}$ are ordered with respect to the rank function defined below. The k'' will then be chosen in that order, the index with the smallest rank first, until the next "new" alarm.

The rank function is determined on the basis of two criteria: first, for reasonable sample size N the error probability for a single decision is small and thus the (wrong) decision for the first false estimator should be "close". To quantify this idea, let absdiff(i) denote the number of indices s in $\{i+8,\ldots,i+1\}$ for which the absolute value $|\psi_{N,d} - \mathit{decbound}|$ is smaller than the respective term for index i. Here *decbound* corresponds to the decision boundary, marking the limit between a decision towards 0 and towards 1 in the decision strategy, and is defined as $\log(\frac{m-1}{k-m+1}) + \frac{1}{2}\sum_{j=1}^{N} \log(1+c_j)$ (cf. Theorem 1). Intuitively, absdiff(i) denotes the number of recent (relative to i) positions for which the decision was more difficult to make than for index i.

Second, from the $\psi_{N,d}(\cdot)$-values from which the estimators $\widetilde{d}_{con-1},\ldots,\widetilde{d}_{k+1}$ were derived we guess the "region" where the first false estimator is located. For this, we define the intervals $I_1 := (-\infty, E_{\theta=1}(Z)]$, $I_2 := (E_{\theta=1}(Z), \mathit{decbound}]$, $I_3 := (\mathit{decbound}, E_{\theta=0}(Z)]$, and $I_4 := (E_{\theta=0}(Z), \infty)$ (these regions can be easily visualized on Fig. 1). For $s \in \{\mathrm{con}-1,\ldots,k+1\}$ we set $iv(s) := j$ if the respective $\psi_{N,d}(\cdot)$-value was contained in I_j. Let us now consider the distribution of successive estimators among these regions, and compare these distributions before and after an error. Denote by k^* the first error position:

– before that error, the estimators should be equidistributed among the 4 regions; that is, $\mathrm{Prob(iv(s) = j)} \approx 0.25$ for $s > k^*$;

- at the error position, the probability is high for the estimator to be in I_2 or I_3; in other words, $\mathrm{Prob}(\mathrm{iv}(\mathrm{k}^*) = 2), \mathrm{Prob}(\mathrm{iv}(\mathrm{k}^*) = 3) \approx 0.5$;
- finally, we have seen that, after the error has occurred, decisions will be biased towards 0; the precise distribution is given by Theorem 6.5(iii) in [13], that we omit here for simplicity.

To determine the region where the first error was located, we define $L(t) := 0.25^{\mathrm{con}-t-1} \prod_{s=k+1}^{t} \mathrm{Prob}(\mathrm{iv}(\mathrm{s})))$ and retain as k'' the value of t that maximized this term. (In fact, the terms $\mathrm{Prob}(\mathrm{iv}(\mathrm{s}))$ equal the respective probabilities under the assumption that the first error occurred at index t. The term $L(t)$ is an approximation for the probability for the observed $iv(\cdot)$-values to have occurred if the first error position was t (cf. also [15]).)

Finally, we define posmax: $\{\mathrm{con} - 1, \ldots, k + 1\}$, $\mathrm{posmax}(i) := |k'' - i|$ and use the rank function

$$\mathrm{rank}(i) := 7.0 * \mathrm{absdiff}(i) + 3.0 * \mathrm{posmax}(i) \ . \tag{8}$$

In the same way as for the probabilities p_{err} and p_{len} we used Theorem 6.5(iii) in [13] to pre-compute average values for the probabilities $\mathrm{Prob}(\mathrm{iv}(\mathrm{s}) = \mathrm{j})$ to save computation time. The rank function turned out to be very efficient (cf. Subsect. 3.5).

3.5 Practical Experiments / Efficiency

Although the original attack also used an error correction strategy, about 200000–300000 time measurements were necessary to recover a 512-bit key. In this section we present empirical results where we applied the optimal decision strategy stated in Subsect. 3.2 and the error detection and correction strategy from Subsects. 3.3 and 3.4. For our experiments we distinguished two cases. In the *ideal case* we assumed that the time measurements are exact, the attacker knows the constants and parameters c, c_{ER}, w and $\mathrm{h}am(d)$ and he is able to determine the setup time $t_{(S)}$ exactly. (This corresponds to a computer emulation of Algorithm 1 with the number of extra reductions as output.) However, if the attacker does not have exact knowledge of the smart card implementation these assumptions may be not realistic. Therefore, we also conducted a suggestively called *real-life attack* where the attacker's knowledge and his abilities were assumed to be lower, using actual timing measurements from a smart card running a cryptographic library ([2]). In this case, we assumed that the attacker does not have precise implementation knowledge and exploited various relations to estimate these values. We refer the interested reader to [15] (Sect. 6) for further details.

Remark 4. As no physical smart card is available yet, timing measurements were in fact performed using an emulator [1] which predicts the running time (in clock cycles) of a program. The code we used was the ready-for-transfer version of the Cascade library, i.e. with critical routines directly written in the card's native assemble language. Since the emulator is designed to allow implementors to optimize their code before "burning" the actual smart cards, its predictions

should match almost perfectly. Consequently, physical attacks on the smart card should not induce many measurement errors more.

Tables 1 and 2 contain empirical results where we used the optimal decision strategy without (Table 1) and combined with error detection and correction (Table 2). The last two columns respectively correspond to an ideal case, in which all $t_{d(j)}$ are measured exactly, and to the real-life attack of the Cascade chip.

Table 1. Optimal decision strategy without error correction

Key size (bits)	number of measurements	Success rate ideal case	Success rate real-life attack
512	5 000	12%	15%
512	6 000	35%	32%
512	7 000	55%	40%
512	8 000	65%	46%
512	9 000	95%	72%
512	10 000	98%	92%

Table 2. Optimal decision strategy with error correction

Key size (bits)	number of measurements	Success rate ideal case	Success rate real-life attack
512	5 000	85%	74%
512	6 000	95%	85%
512	7 000	98%	89%
512	8 000	100%	91%
512	9 000	100%	94%
512	10 000	100%	100%

Compared with the original attack presented in [6], our approach – optimal decision strategy combined with the error detection and correction strategy described below – has improved the efficiency by a factor of about 50 for 512-bit keys, at no cost in prerequisites, generality, or complexity. This improvement factor is expected to grow even further with larger keys. The main portion of the improvement is obviously due to the optimal bit estimation strategy but a brief comparison between both tables shows that the applied error handling itself reduces the sample size by about a 40 per cent. In the ideal case in more than 70 per cent of the trials the index of the first false decision was ranked on position one or two. In the real-life attack the efficiency of the rank function is somewhat lower.

To conclude this section, an important point to note is that, in this case, the detection and correction of errors does not cost any additional time measurement.

4 Timing Attack against RSA with CRT

4.1 Description of the attack

Let $n = p_1 p_2$ denote an RSA modulus where p_1 and p_2 are large primes. If the CRT is used the computation $y \mapsto y^d (\mathrm{mod}\, n)$ decomposes into three steps:

Step 1: $y_1 := y (\mathrm{mod}\, p_1)$. Compute $x_1 := (y_1)^{d'} (\mathrm{mod}\, p_1)$
Step 2: $y_2 := y (\mathrm{mod}\, p_2)$. Compute $x_2 := (y_2)^{d''} (\mathrm{mod}\, p_2)$
Step 3: Return $(b_1 x_1 + b_2 x_2) (\mathrm{mod}\, n)$

The parameters d', d'', b_1 and b_2 are precomputed once. In particular, $d' := d (\mathrm{mod}\,(p_1 - 1))$, $d'' := d (\mathrm{mod}\,(p_2 - 1))$ while $b_1 \equiv 1 \pmod{p_1}$ and $b_1 \equiv 0 \pmod{p_2}$, and similarly, $b_2 \equiv 0 \pmod{p_1}$ and $b_2 \equiv 1 \pmod{p_2}$. Unlike as in Sect. 3 the attacker knows neither the bases y_i nor the moduli p_i in Steps 1 and 2. In particular, the "classical" timing attacks ([7], [6], [13]) do not work. In [14] however, a new timing attack was introduced which enables the factorization of n if the modular multiplications $(\mathrm{mod}\ p_i)$ are carried out with Montgomery's algorithm. We briefly describe the attack. For details, the interested reader is referred to [14].

As the primes p_1 and p_2 are of similar size it is reasonable to assume the Montgomery constant R (cf. Sect. 3) is equal for both multiplications $(\mathrm{mod}\ p_1)$ and $(\mathrm{mod}\, p_2)$. For simplicity, for the moment we assume that the modular exponentiations in Steps 1 and 2 are carried out with the square & multiply algorithm. In accordance to Sect. 3 we define the mappings $\Psi_i \colon Z \to Z_{p_i}$ by $\Psi_i(z) := zR\, (\mathrm{mod}\, p_i)$. Let $0 < u_1 < u_2 < n$ with $u_2 - u_1 < p_1, p_2$. Three cases are possible:

Case A: $\{u_1 + 1, \ldots, u_2\}$ does not contain a multiple of p_1 or p_2.
Case B: $\{u_1 + 1, \ldots, u_2\}$ contains a multiple of one of p_1 or p_2 but not of both.
Case C: $\{u_1 + 1, \ldots, u_2\}$ contains a multiple of both p_1 or p_2.

Let R^{-1} denote the multiplicative inverse of R modulo n. For input $uR^{-1} (\mathrm{mod}\ n)$ clearly $\Psi_i(uR^{-1}\, (\mathrm{mod}\, n)) = u\, (\mathrm{mod}\, p_i)$ for $i = 1, 2$. In Step i the probability for an extra reduction in a Montgomery multiplication with $u\, (\mathrm{mod}\, p_i)$ equals $(u\, (\mathrm{mod}\ p_i)/2R)$ (cf. (3) or Theorem 1 in [14]) while the probability for an extra reduction in a squaring is $p_i/3R$, independent of the base. The running time for the input $uR^{-1}\, (\mathrm{mod}\, n)$, denoted with $T(u)$, is interpreted as a realization of a random variable X_u (cf. [14]). The expectation of the difference $X_{u_2} - X_{u_1}$ depends essentially on the fact whether Case A, Case B or Case C is true:

$$E(X_{u_2} - X_{u_1}) \approx \begin{cases} 0 & \text{in Case A} \\ -\frac{c_{\mathrm{ER}}}{8} \frac{\sqrt{n}}{R} & \text{in Case B} \\ -\frac{c_{\mathrm{ER}}}{8} \frac{2\sqrt{n}}{R} & \text{in Case C.} \end{cases} \tag{9}$$

This observation is essential for the attack which falls in three phases: In Phase 1 an "interval set" $\{u_1 + 1, \ldots, u_2\}$ has to be found which contains an integer multiple of p_1 or p_2. Starting from this set, in Phase 2 a sequence of decreasing interval subsets has to be determined, each of which containing an integer multiple of p_1 or p_2. More precisely, in each step of Phase 2 it is checked whether the upper subset $\{u_3 + 1, \ldots, u_2\}$ with $u_3 := \lceil (u_1 + u_2)/2 \rceil$) contains such a multiple or not. The decisions in Phase 1 and 2 are based on the time differences $T(u_2) - T(u_1)$ or $T(u_2) - T(u_3)$, resp., where the attacker decides for "Case A" iff $T(u_2) - T(u_1) > -c_{\text{ER}}\sqrt{n}/16R$ or $T(u_2) - T(u_3) > -c_{\text{ER}}\sqrt{n}/16R$, resp. (Note that there is no need to distinguish between Cases B and C.) When the actual subset $\{u_1+1, \ldots, u_2\}$ is sufficiently small Phase 3 begins where $\gcd(u, n)$ is calculated for all u contained in this subset. If all decisions within Phase 1 and 2 were correct then the final subset indeed contains a multiple of p_1 or p_2. Then Phase 3 delivers the factorization of n.

4.2 Detecting errors

However, at any instant within Phase 2 the attacker can verify with high probability whether his decisions were correct so far, i.e. whether the actual interval $\{u_1 + 1, \ldots, u_2\}$ really contains a multiple of p_1 or p_2. He just has to apply the decision rule to a time difference for neighbouring values of u_1 and u_2, resp., e.g. to $T(u_2 - 1) - T(u_1 + 1)$. If this leads to the same decision it is confirmed with overwhelming probability that the interval $\{u_1 + 1, \ldots, u_2\}$ truly contains a multiple of p_1 or p_2. (We then call $\{u_1 + 1, \ldots, u_2\}$ a *confirmed interval.*)

Otherwise, the attacker evaluates a further time difference (e.g. $T(u_2 - 2) - T(u_1 + 2)$). Depending on this difference he either finally confirms the interval $\{u_1+1, \ldots, u_2\}$ or restarts the attack at the preceding confirmed interval $\{u_{1;c}+1, \ldots, u_{2;c}\}$ using values u_1' and u_2' close to $u_{1;c}$ and $u_{2;c}$, resp.

As opposed to the attack of section 3, this error detection method requires additional time measurements to be carried out, and has therefore significant impact on the attack's efficiency. It would thus be useful to be able to reduce the number of detection steps applied.

In [14] a static error detection and correction is applied: After a pre-assigned number of steps (adapted to the parameters n and R) a new confirmed interval is tried to be established. If this fails (due to a preceding error) the attack is restarted at the preceding confirmed interval. For $n \approx 0.7 \cdot 2^{1024}$ and $R = 2^{512}$ practical attacks required 570 time measurements in average where confirmed intervals were tried to establish after each 42 steps.

Remark 5. (i) This attack is somewhat atypical as it does not directly reconstruct the secret exponent d itself. Instead, the interval sequence delivers the bit representation of a multiple of p_i, beginning with the most significant bits. This multiple, however, may be viewed as a "key" as its knowledge enables the factorization of the modulus.
(ii) If the the initial value u_2 in Phase 1 is chosen sufficiently small the attacker

will find a prime factor p_i itself rather than just a multiple of it. Using an algorithm of Coppersmith (cf. [3]), it then suffices to reconstruct the upper half of the bit representation of p_i which almost halves the number of time measurements in Phase 2. For $n \approx 0.7 \cdot 2^{1024}$ and $R = 2^{512}$, for example, about 300 time measurements are sufficient for the whole attack (cf. [14] (Remark 6)).
(iii) Although its efficiency decreases the attack also works if table methods are used in Steps 1 and 2 of the CRT. For a 4-ary table (storing $2^4 - 1 = 15$ values), for example, about 17700 time measurements are required in average ([14], Sect. 7).

4.3 Dynamic Error Detection and Correction

However, this static error detection and correction strategy can still be improved, using a similar argument to the one of section 3.3. In fact, after a wrong decision in Phase 2 the following decisions should always be "no multiple of p_1 or p_2 in the upper subset $\{u_3 + 1, \ldots, u_2\}$". If all decisions were correct so far, however, within Phase 2 this event should occur with probability 1/2.

This suggests the following error detection and correction strategy: If in none of the preceding v (let's say $v = 13$) steps a multiple of p_1 or p_2 was assumed in the respective upper subinterval then try to confirm the actual interval $\{u_1 + 1, \ldots, u_2\}$ by evaluating $T(u_2 + 1) - T(u_1 + 1)$. If this attempt fails, restart the attack at the last but one interval where a multiple of p_1 or p_2 was assumed in the upper subset (restart with neighbored values).

For $n \approx 0.7 \cdot 2^{1024}$ and $R = 2^{512}$, for example, the static strategy described above requires about 45 of 570 time measurements in average for error detection or correction reasons. However, the probability for a single decision in Phase 1 or 2 to be wrong is about 0.001. Hence about 0.5 errors are expected within the attack whereas the probability for a "false alarm" is about $2^{8-13} = 1/16$. Thus the proposed new error detection and correction strategy costs about $0.5(2+15+2) + 2/32 \approx 10$ time measurements which reduces the average number of time measurements to about 535 which is a reduction by 6 per cent. If the modular exponentiations in Steps 1 and 2 of the CRT use tables, then the portion of time measurements carried out due to error detection and correction reasons is considerably larger than for the square & multiply algorithm. Consequently, the gain of efficiency caused by a dynamic error detection and correction strategy (with an adapted value v) also increases.

5 Timing Attack against AES (Rijndael)

5.1 Brief Description of Rijndael and the Vulnerable Model

A complete description of Rijndael can be found in [4]. We will focus here on the parts of interest for the attack.

A Rijndael encryption consists in an initial round key addition, followed by M round transformations, the last round being slightly different from the others.

The different transformations applied during each round operate on an array of bytes, named the *state*, composed of 4 lines and M_b columns (where M_b is the block size, in 32-bit words). Basically, each round, except the last one, consists of the following steps: `ByteSub` (byte-by-byte substitution S), `ShiftRow` (fixed permutation of bytes), `MixColumn` (described below) and `AddRoundKey` (round key $\oplus$ state) where "$\oplus$" denotes the bitwise XOR-addition. One `AddRoundKey` operation is performed before the first round. In the final round there is no `MixColumn` operation.

The `MixColumn` transformation operates on the columns of the state and applies them the following matrix multiplication:

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{10}$$

where the multiplication is defined in $GF(2^8)$ as multiplication of binary polynomials modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.

Due to the choice of the matrix and irreducible polynomial, `MixColumn` can be implemented very efficiently: first, it is easy to see that, as '03'='02'+'01', the only multiplications that will actually have to be performed are by '02'; second, it can be showed (cf. [4]) that multiplication by '02' in GF(256) can be implemented as follows:

– shift byte one position left,
– if a carry occurs, XOR the result with hexadecimal '1B'.

Assumptions. If the implementation is careless, the multiplications with '02' and '03' will not take constant time, but will take longer in the case where a carry occurs. Throughout the rest of this section, we assume such a behaviour. Moreover, we assume that the time needed for the remaining operations, i.e. for substitutions, permutations and XOR additions, does not depend on the plaintext and thus is constant for all encryptions. (However, slight deviations could be interpreted as a part of the measurement error; cf. Subsection 5.3.) For simplicity, we further assume that the length of the key ($=F$ bytes) is not larger than that of a plaintext block ($=4M_b$ bytes). Finally, we assume that the attacker knows at least one pair $(p, c := \mathrm{E}nc(p;k))$ where $\mathrm{Enc}(\cdot;k)$ stands for a Rijndael encryption with the secret (unknown) key k. This pair will be used to check whether a key candidate is correct or not.

5.2 Basic Idea

Let us focus on what will happen to a given byte (say, the first) of a known plaintext during the first few encryption sub-steps:

– before entering the first round, that byte will be XOR-ed with the first byte of the first round key (equals the first byte of k) – call it R_1 – whose value is unknown, but constant independent of the plain text;

– then a substitution S, according to a known S-box, will take place;
– the byte will then be moved around (to a known place) by `ShiftRow`, without being modified;
– finally, `MixColumn` will be applied; during this operation, the byte will be multiplied at least once by '02' (the result may be stored and reused for the multiplication with '03'). By assumption the multiplications will take longer if the first bit of the byte is set.

Could we observe the time taken by that peculiar multiplications, we would then be able to deduce the value of R_1 with about 8 encryptions. Of course, this is not the case, as we have no access to partial timings, but only to the total encryption time. However, if we encrypt a large amount of random messages, we can expect all other operations to behave as random noise and therefore hope to be able to "filter out" the information we are interested in. Naturally, the same method can be applied to guess the other bytes $R_2, \ldots, R_F$ of the searched secret key k.

To make the attack robust against errors induced by random noise, we will not build a single answer, but rather a set of possible keys, small enough to make exhaustive search easy. More precisely, for each index $j \leq F$ the attacker determines a (small) set of "candidates", denoted with Ca_j. After the "guessing phase" has been finished the attacker computes $\mathrm{E}nc(p; ca_1 \| ca_2 \| \cdots \| ca_F)$ for all possible combinations of key byte candidates (i.e. $ca_j \in Ca_j$ for each $j \leq F$). If $\mathrm{E}nc(p; ca_{1;0} \| ca_{2;0} \| \cdots \| ca_{F;0}) = c$ then $k = ca_{1;0} \| ca_{2;0} \| \cdots \| ca_{F;0}$ with overwhelming probability.

5.3 The Pure Attack

We begin with a definition.

Definition 1. *The Greek letter Δ denotes the extra time needed for the multiplications of a byte with '02' and '03' if a shift occurs. The term $N(\mu, \sigma^2)$ denotes a normal distribution with mean μ and variance σ^2. In particular, $f(t) := e^{-(t-\mu)^2/2\sigma^2}/\sqrt{2\pi}\sigma$ denotes its Lebesgue density.*

In the first step the attacker randomly chooses plaintexts $p_1, \ldots, p_N$ and measures the running times $\widetilde{t}_1, \ldots, \widetilde{t}_N$ needed for their encryption with the secret key k. In particular, $\widetilde{t}_j = t_j + t_{\mathrm{E}rr}$ with $t_j = \mathrm{T}ime(\mathrm{E}nc(p_j; k))$ while $t_{\mathrm{E}rr}$ denotes the measurement error. Then he initializes tables $U_1, \ldots, U_F$ where $U_s := (u_s[i][j])_{0 \leq i \leq 255; 1 \leq j \leq N}$. The component $u_s[i][j]$ equals the most significant bit of $S(i \oplus s^{th}$ byte of $p_j)$. In other words, $u_s[i][j]$ tells – assuming the sth byte of the key, R_s, is equal to i – whether the first multiplication by '02' applied to the corresponding byte involves a shift or not.

If $R_s \neq i$ the measured running time $\widetilde{t}_j$ can be interpreted as a realization of a normally distributed random variable $\widetilde{T}_j \sim N(\mu, \sigma^2)$ where $\sigma^2 = \sigma_\Delta^2 + \sigma_{\mathrm{Err}}^2$ with $\sigma_\Delta^2 := 4M_b(M-1)\Delta^2/4$ while σ_{Err}^2 denotes the variance of the measurement error. However, if $R_s = i$ the table entry $u_s[i][j]$ provides an additional

piece of information (concerning particular multiplications by '02' and '03'). Consequently, $\widetilde{T}_j \sim N(\mu + \Delta(u_s[i][j] - 0.5), \sigma^2 - \Delta^2/4)$. The original question, namely whether $R_s = i$, can be viewed as a more general problem which is independent of any cryptographic context: namely, whether $\widetilde{T}_j \sim N(\mu, \sigma)$ or $\widetilde{T}_j \sim N(\mu + \Delta(u_s[i][j] - 0.5), \sigma^2 - \Delta^2/4)$ for $j \le N$. The measured running times $\widetilde{t}_1, \ldots, \widetilde{t}_N$ is the only information available. For this, statistical decision theory can be applied.

Remark 6. (parameter estimation) For simplicity, first assume that the attacker is able to use a device identical to the one he wants to attack with a known key. He then randomly generates plaintexts $p'_1, \ldots, p'_{N1}$ and $p''_1, \ldots, p''_{N1}$ with constant first byte in each subset such that *for the known key* for the respective multiplications by '02' and '03' a shift occurs (subset 1), resp. does not occur (subset 2). The difference in the mean values delivers an estimator for Δ (and thus for σ^2_Δ) and their arithmetical mean an estimator for μ. If the attacker cannot use a known key the estimation strategy is similar; he just has to identify two subsets with the properties from above, which can easily be done as follows: he starts by building two subsets of plaintexts (A, B), in which the first byte of every element of A and the first byte of every element of B are fixed to (two different) constants. If the average time for processing subset A significantly differs from that for subset B, then the attacker has identified the two desired sets; otherwise he simply repeats the operation with a third subset, comparing its average processing time with that of the second etc.

Roughly speaking, in a statistical decision problem the statistician (here: the attacker) estimates the unknown distribution of random variable(s), p_θ, on the basis of an observation $\omega \in \Omega$. The set Θ describes all possible alternatives. We will not consider statistical decision problems in full generality (cf., e.g. [17]) but apply the mechanisms to our specific problem. Here the parameter set Θ equals $\{0, 1\}$ where $\theta = 0$ denotes the case $i = R_s$ while $\theta = 1$ stands for $i \neq R_s$. The probability that for byte i the hypothesis $\theta = 0$ (resp., $\theta = 1$) is true equals $\eta_0 = 1/256$ (resp., $\eta_1 = 255/256$). If the attacker decides for $\theta = 0$ although $\theta = 1$ is true he needlessly adds one candidate i to Ca_s which increases the time for final exhaustive search. Erroneously deciding for $\theta = 1$, i.e. cancelling the true byte value R_s, is much worse as the attack must fail in the end. To quantify these considerations, we introduce the loss function $s(\theta, \theta') \ge 0$ where the first component denotes the correct parameter and the second the estimated one. Of course, $s(0, 0) = s(1, 1) = 0$ (correct decisions). We further set[4] $s(1, 0) = 1$ and $s(0, 1) = 100$. Of course, the attacker uses the decision strategy d_{opt} which minimizes the expected loss, that is, $d_{opt}(\widetilde{t}_1, \ldots, \widetilde{t}_N) := u \in \{0, 1\}$ if u minimizes the term

$$\sum_{\theta=0}^{1} \eta_\theta s(\theta, u) \prod_{j=1}^{N} f_{\theta;j}(\widetilde{t}_j). \tag{11}$$

[4] The ratio $s(0, 1)/s(1, 0)$ is somewhat arbitrary. Increasing $s(0, 1)$ reduces the probability that the correct value is cancelled but simultaneously increases the candidate set Ca_s.

For the moment, $f_{1;j}$ (resp., $f_{0;j}$) denotes a normal density with mean μ and variance σ^2 (resp., with mean $\mu + \Delta(u_s[i][j] - 0.5)$ and variance $\sigma^2 - \Delta^2/4$). As the variances for $\theta = 0$ and $\theta = 1$ are almost equal the attacker may use the simplified decision strategy $d'_{opt}(\widetilde{t}_1, \ldots, \widetilde{t}_N) = 0$ instead iff

$$\sum_{j=1}^{N} \left((\mu + \Delta(u_s[i][j] - 0.5) - \widetilde{t}_j)^2 - (\mu - \widetilde{t}_j)^2 \right) < 2\sigma^2(\log(s(0,1) - \log(255)) \tag{12}$$

with only minimal loss of efficiency. The attacker applies this decision rule to each $i \in \{0, \ldots, 255\}$ which yields the candidate set Ca_s.

5.4 Error Location and New Guesses

This timing attack on Rijndael is an atypical divide and conquer attack as a wrong key estimator does not influence the following partial key guesses. In principle, the attacker could thus apply the decision strategy from the preceding section independently to all key bytes and completely resign on any intermediate error detection strategy. On the other hand, the candidate sets must be kept small enough for the verification phase to remain practically feasible. We propose the following strategy.

We assume that the attacker has already determined the candidate sets $Ca_1, \ldots, Ca_{s-1}$. To derive Ca_s we initialize a matrix $(c_m, i)_{1 \leq m \leq |Ca_{s-1}|; 0 \leq i \leq 255}$ where $c_m \in Ca_{s-1}$. Our aim is to "tick" all components for which at least one component is a correct candidate (for R_{s-1} or R_s, resp.). If this can be managed with reasonable probability, many ticks should occur in the row indexed by the correct candidate $c_x = R_{s-1}$ and in the column indexed by $i = R_s$. This suggests the following strategy: cancel all elements of Ca_{s-1} besides y (let's say $y \leq 4$) candidates whose rows had the most ticks; similarly, build a set Ca_s containing z (let's say $z \leq 20$) bytes for which in the corresponding columns the most ticks occur.

An efficient "ticking strategy" remains to be derived. For each component (c_m, i), four cases are possible: Case A: $(c_m = R_{s-1}, i = R_s)$, Case B: $(c_m = R_{s-1}, i \neq R_s)$, Case C: $(c_m \neq R_{s-1}, i = R_s)$ and Case D: $(c_m \neq R_{s-1}, i \neq R_s)$. Then $\widetilde{T}_j$ is normally distributed with mean $\mu + \Delta(u_{s-1}[c_m][j] + u_s[i][j] - 1.0)$, $\mu + \Delta(u_{s-1}[c_m][j] - 0.5)$, $\mu + \Delta(u_s[i][j] - 0.5)$ or μ, resp., and variance $\sigma^2 - \Delta^2/2$, $\sigma^2 - \Delta^2/4$, $\sigma^2 - \Delta^2/4$, or σ^2, resp. For the moment the respective densities are suggestively denoted with $f_{A;j}$, $f_{B;j}$, $f_{C;j}$, $f_{D;j}$. The cases A, B, C and D occur with probabilities $\eta_A = 1/256|Ca_{s-1}|$, $\eta_B = 255/256|Ca_{s-1}|$, $\eta_C = (|Ca_{s-1}| - 1)/256|Ca_{s-1}|$ and $\eta_D = 1 - \eta_A - \eta_B - \eta_C$. If Case D is true, the loss for yet ticking the component (c_m, i) is set $s_D := 1$. If Case B or C is true, but the component is not ticked, the loss equals $s_{BC} := 30$ whereas $s_A := 2s_{BC}$ for Case A. (Recall that we are only interested in the total number of ticks in a row or column but not in their positions.) This leads to the following decision rule:

Tick the component (c_m, i) iff

$$\eta_D s_D \prod_{j=1}^{N} f_{D;j}(\widetilde{t}_j) < \eta_A s_A \prod_{j=1}^{N} f_{A;j}(\widetilde{t}_j) + s_{BC}\left(\eta_B \prod_{j=1}^{N} f_{B;j}(\widetilde{t}_j) + \eta_C \prod_{j=1}^{N} f_{C;j}(\widetilde{t}_j)\right). \tag{13}$$

As the attack itself, its error handling is also atypical. There is no error correction but some candidates (which are assumed to be false) are cancelled. In the original meaning of the word there is no error detection either, as errors must trivially have occured if the previous candidate set contains more than one element. However, some false estimators are located and cancelled. The location of previous errors and the estimation of new key byte candidates are done simultaneously. Compared with the pure attack, the number of operations in the guessing phase increases linearly with the average size of the candidate sets before their reduction. On the other hand, the number of operations in the verifying phase decreases by an (exponential) factor of about β^F where β denotes the average ratio between the size of a candidate set *after* cancelling wrong candidates and the size *before* the cancelling. (Note that Ca_F can finally be reduced by applying error location to $Ca_F \times \{0, \ldots, 255\}$.) Consequently, before the cancelling, the candidate sets may be much larger than for the pure attack, which means that the number of time measurements can be reduced significantly.

5.5 Practical Results

In [8] an attack was experimented against the 128-bit block, 128-bit key version of Rijndael with $M = 10$ rounds. It turned out that, with 3000 samples per key byte, the complete key was recovered with very high probability at negligible cost. The attack described above reduced the total number of time measurements from $16 \cdot 3000 = 48000$ to 4000, with a success rate of more than 90%. The set Ca_1 was determined with the pure attack, $Ca_2, \ldots, Ca_F$ with the strategy described in the preceding subsection. The same time measurements were used for all key bytes. The decisions within the pure attack and the error location strategy for $s = 2$ were strongly correlated. Hence for $s = 2$ we used $Ca_1' := \{0, \ldots, 255\}$ instead of Ca_1. A more time-efficient variant which yet requires more time measurements is to use another sample of size 2000 for the pure attack.

Note that the sets Ca_i are in fact ordered, with the most probable candidates first. Therefore the final "exhaustive search" phase generally succeeds after exploring a very small portion of the search space.

Although the new attack is much more efficient than the one in [8] both attacks are based on the same basic idea (cf. Subsect. 5.2) and exploit the same implementation weakness.

6 Algorithmically Based Divide and Conquer Attacks

An important design criterion for cryptographic algorithms is that it must not be possible to confirm or validate partial key guesses, with reasonable proba-

bility, using algorithmically based attacks, i.e. attacks which do not consider implementation details. Divide and conquer attacks thus typically occur in side channel attacks. However, they are not restricted to that field. Siegenthaler's attack (cf. [16]), for example, has been well-known for almost 20 years.

In [16] Siegenthaler analyzes a divide and conquer attack on a particular class of stream ciphers for which the key stream $k_1, k_2, \ldots$ (single bits) is generated by m linear feedback shift registers $LFSR_1, \ldots, LFSR_m$ of length $r_1, \ldots, r_m$ with primitive feedback polynomials $q_1, \ldots, q_m$ whose output values at time t, denoted with $x_{1(t)}, \ldots, x_{m(t)}$, are memoryless combined with $f\colon \{0,1\}^m \to \{0,1\}$. More precisely, $k_t := f(x_{(1)t}, \ldots, x_{(m)t})$ and $c_t := p_t \oplus k_t$ where p_t and c_t denote the tth plaintext and ciphertext bit, resp. The searched key is the initial values of $LFSR_1, \ldots, LFSR_m$ and, if the feedback polynomials are unknown, also of q_i. Let the random variables $X_1, \ldots, X_m$ be independent and equidistributed on $\{0,1\}^{r_1}, \ldots, \{0,1\}^{r_m}$, resp. If $\epsilon_i := \mathrm{Prob}(\mathrm{X_i} = \mathrm{f}(\mathrm{X_1}, \ldots, \mathrm{X_m})) - 0.5 \neq 0$, the ith part of the key (the initial value of $LFSR_i$ and eventually q_i) may be guessed independently from the remainder. More precisely, if the candidate for the ith key part is correct then $\#\{1 \le t \le N \mid x_{i(t)} = f(x_{1(t)}, \ldots, x_{m(t)})\} \approx (0.5 + \epsilon_i)N$ can be expected whereas $\approx 0.5N$ else. If $\epsilon_i \neq 0$ for all i, this enables a straight-forward divide and conquer attack with known plaintext. In fact, even a ciphertext-only-attack is feasible (cf. [16]). Vice versa, the use of correlation immune combiners of high order (eventually with memory) prevents such attacks (cf., e.g., [16], [12]) as the attacker then had to guess many key parts simultaneously which is practically infeasible.

As in the timing attack on Rijndael but unlike in the other attacks treated in this paper the key parts can be guessed independently. If the available sample size N is small (in relation to the ϵ_i), for each key part so many candidates may remain that the verifying phase may be very costly. Depending on the concrete combiner f, however, there may exist a very obvious and near at hand error location strategy. Assume, for example, that the absolute value $|\epsilon_{j_1,\ldots,j_b}| := |\mathrm{Prob}(\mathrm{X_{j_1}} \oplus \cdots \oplus \mathrm{X_{j_b}} = \mathrm{f}(\mathrm{X_1}, \ldots, \mathrm{X_m})) - 0.5|$ is fairly large. Then we can pre-check the cartesian product $Ca_{j_1} \times \cdots \times Ca_{j_b}$ of candidate sets. However, unlike in the timing attack against Rijndael, there is no tendency for "ticks" in particular regions of $Ca_{j_1} \times \cdots \times Ca_{j_b}$. More precisely, we can expect correlations only for the b-tuple for which all components are correct; other pre-confirmed b-tuples result from statistical deviations. We hence do not pre-confirm subsets of $Ca_{j_1}, \ldots, Ca_{j_b}$ but a subset of $Ca_{j_1} \times \cdots \times Ca_{j_b}$ whose elements are the remaining candidates for $LFSR_{j_1}, \ldots, LFSR_{j_b}$.

7 Conclusion

This paper proposed a method to improve the error detection/correction step in divide and conquer attacks, independently of the attack itself. Although the method is not a "ready-to-use" tool, in the sense that some work and analysis is necessary to instantiate it to a particular divide and conquer problem, we believe

the above examples have highlighted its basic principles, allowing a motivated user to apply it with reasonable effort.

Moreover, the efficiency of an adequate error management policy is often big enough (sometimes it reduces the necessary sample size up to factor 2) to make it worth this effort, especially in a real-world attack.

A good understanding of the potential power of an attack is necessary to be able to design adequate countermeasures. This paper, focusing on an in-depth analysis of the available data in order to exploit them in a nearly-optimal way, attempted to give an insight of that potential power.

References

1. *ARM Software Development Toolkit version 2.11.* Advanced RISC Machines Ltd, 1997. User's guide document number: ARM DUI 0040C.
2. Cascade (Chip Architecture for Smart CArds and portable intelligent DEvices). Project funded by the European Community, see http://www.dice.ucl.ac.be/crypto/cascade.
3. D. Coppersmith: Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. J. Cryptology **10** (no. 4) (1997) 233–260.
4. J. Daemen, V. Rijmen: AES proposal: Rijndael. In: Proc. first AES conference, August 1998. Available on-line from the official AES page: `http://csrc.nist.gov/encryption/aes/aes_home.htm`.
5. J.F. Dhem.: Design of an Efficient Public-Key Cryptographic Library for RISC-Based Smart Cards. PhD thesis, Université catholique de Louvain - UCL Crypto Group - Laboratoire de microélectronique (DICE), May 1998.
6. J.-F. Dhem, F. Koeune, P.-A. Leroux, P.-A. Mestré, J.-J. Quisquater, J.-L. Willems: A Practical Implementation of the Timing Attack. In: J.-J. Quisquater and B. Schneier (eds.): Smart Card – Research and Applications, Springer, Lecture Notes in Computer Science, Vol. **1820**, Berlin (2000), 175–191.
7. P. Kocher: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems. In: N. Koblitz (ed.): Advances in Cryptology – Crypto '96, Springer, Lecture Notes in Computer Science **1109**, Berlin (1996), 104–113.
8. F. Koeune, J.-J. Quisquater: A Timing Attack against Rijndael. Université catholique de Louvain, Crypto Group, Technical report CG-1999/1, 1999.
9. A.J. Menezes, P.C. van Oorschot, and S.C. Vanstone: Handbook of Applied Cryptography, Boca Raton, CRC Press (1997).
10. T.S. Messerges, E.A. Dabbish, R.H. Sloan: Power Analysis Attacks of Modular Exponentiation in Smartcards. In: Ç.K. Koç, C. Paar (eds.): Cryptographic Hardware and Embedded Systems — CHES 1999, Springer, Lecture Notes in Computer Science, Vol. **1717**, Berlin (1999), 144–157.
11. P.L. Montgomery: Modular Multiplication without Trial Division, Math. Comp. **44**, no. 170, 519–521 (April 1985).
12. R.A. Rueppel: Analysis and Design of Stream Ciphers, Springer, Berlin (1986).
13. W. Schindler: Optimized Timing Attacks against Public Key Cryptosystems. To appear in Statistics & Decisions.
14. W. Schindler: A Timing Attack against RSA with the Chinese Remainder Theorem. In: Ç.K. Koç, C. Paar (eds.): Cryptographic Hardware and Embedded Systems — CHES 2000, Springer, Lecture Notes in Computer Science **1965**, Berlin (2000), 110–125.

15. W. Schindler, F. Koeune, J.-J. Quisquater: Unleashing the Full Power of Timing Attacks. Université catholique de Louvain, Crypto Group, Technical report CG-2001/3, 2001.
16. T. Siegenthaler: Decrypting a Class of Stream Ciphers Using Ciphertext Only. IEEE Transactions on Computers. C-34 (1985), 81-85.
17. H. Witting: Mathematische Statistik I, Stuttgart, Teubner (1985).