An Advanced Timing Attack Scheme on RSA

Rudolf Tóth, Zoltán Faigl, Máté Szalay and Sándor Imre
Department of Telecommunications
Budapest University of Technology and Economics
Email: toth.rudolph@gmail.com, zfaigl@mik.bme.hu
szalay@mlabdial.hit.bme.hu and imre@hit.bme.hu

***Abstract*—This paper describes an advanced timing attack scheme on cryptographic algorithms. An attacker can use our method to break a cryptographic algorithm by reconstructing the secret key. The paper contains a detailed explanation of our novel algorithm, furthermore, a practical example for its use. As a proof-of-concept, the method is shown on a specific implementation of the RSA algorithm revealing a 128-bit secret key. Timing attacks assume that the attacker has partial or full knowledge of the internal structure of the attacked algorithm and have gathered time-specific information on a number of known messages, that were encrypted or decrypted with the specific key. In our simplified proof-of-concept example, the attacker knows the total number of extra-reduction steps of the Montgomery multiplication in the RSA for a number of known messages. We demonstrate in practice how this information can be used to achieve complete and fast key recovery with statistical tools, i.e. analysis of variance (ANOVA) and t-test. Similar timing attacks have already been presented by others, however to our knowledge, none of them applied these statistical tools in their methods with such efficiency, and showed the complete recovery in practice by attacking the Montgomery multiplication. However, this is not the main contribution of the paper. The main contribution is, that we have introduced the new concept of key trees and goodness values, which lets the recovery algorithm examine only a very small key space, even if the decision criteria for guessing the key bits are highly biased. This concept can be extended to any other timing attack.**

I. Introduction

Next generation systems are characterized by their openness in terms of allowing and supporting third-party service providers to deploy and compose system-level and application-level services. One major functionality that must be deployed with these services is Authentication, Authorization and Accounting (AAA), furthermore the required level of security and protection must be provided. These functionalities rely on cryptographic algorithms.

Cryptography offers several algorithms that are considered safe on the theoretical level. However, on implementation level there may exist such deceptive signs that expose the algorithms to potential attacks. A timing attack is a side channel attack where the attacker measures and uses time differences between specific events to break a cryptosystem.

There are several papers [1, 2] that present new, or extend existing theoretical timing attacks. There are also some papers [3, 4] which use the results of such theoretical papers to attack some algorithms in practice. Most timing attack algorithms guess the secret key bit by bit. The advantage of this technique is that it is fast, the disadvantage is that if one single bit is missed, it will not find the correct key. Our novel method is a practical one, and overcomes this serious limitation of other timing attacks. We also guess one bit at a time, but not linearly. Basically, our algorithm uses statistical data collected by time measurements to find the secret key. It keeps trying different key prefixes and keys, starting from more probable ones and progressing to less probable ones. The performance depends strongly on how many and how accurate measurements are at the disposal of the attacker. We demonstrate our attack scheme on a specific implementation of the RSA [5] algorithm using Montgomery multiplications [6, 7].

Our most important achievement is the usage of “key trees” and “goodness”. This makes error-correction automatic, and prevents us for searching large subtrees of the keyspace. Besides that, we show in practice how statistical tools, like t-test and ANOVA can be used to retrieve goodness values.

The remainder of the paper is structured as follows. Section II gives an overview of timing attacks, describes the RSA implementation that we are going to use to demonstrate our algorithm, and explains the statistical tools we are going to use. Our attacker-model is introduced in Section III, where it is also explained how the attacker applies the statistical tools to attack the RSA implementation. Section IV presents our novel algorithm in details. It explains how a “key tree” is built, and how “goodness” of various nodes in the tree is computed and used to break the key. Section V presents our proof-of-concept attack against RSA, showing how the algorithm works, and it discusses the performance of the algorithm. Finally, the conclusions are drawn is Section VI.

II. Related Work

A. Timing Attacks

Side channel attacks are a class of cryptographic attacks where the attacker (often referred to as Alice in the litterature) uses information leaked over a channel different from the main output of the system. The attacker uses not only ciphertext blocks or plaintext-ciphertext pairs to break an algorithm, but also measures and uses some other properties of the system.

This special property might include the CPU load or power consumption of the system (or part of the system) executing the cryptographic operations, unintentional electromagnetic radiation caused by various components, or the time a specific cryptographic operation takes to execute.

A timing attack is a side channel attack where the attacker measures and uses time differences between specific events in the system [1]. It is a very important class of the side channel attacks that sometimes can even be carried out remotely.

Take block ciphers as an example. Knowing the internal structure and the implementation details of a specific block cipher, the attacker measures the time it takes to encrypt (or decrypt) a specific known (or chosen) block. The attacker knows everything except for the key used for the cryptographic operations (Kerckhoffs' principle [8]). Based on the measurements, the attacker makes assumptions on the key or parts of the key. These assumptions may be weak and may hold only with a certain probability, or the attacker may make an assumption based on only a small fraction of the measurements she makes. However, the more measurements the attacker may make, the more information she will have on the key. Once enough information is available, a brute force attack may become feasible. The more accurate the measurements of the attacker are, the stronger assumptions can be made.

The idea of timing attacks is not new [1, 9]. These papers focus on the theoretical possibilities of the timing attack against specific implementations of algorithms, and discuss possible countermeasures to make the implementation resist this kind of attack. We, on the other hand, present a practical method, how a timing attack can be mounted against various algorithms.

Dhem et. al. [3] also present a practical attack against RSA. The way RSA is attacked is slightly different from our method presented in Section V. However, this is not the most important difference. While Dhem et. al. write that "We tried several of the tools that statistics offers to compare two samples, such as the Chi-square, Student, Hotteling, and even a test from non-parametric statistics, the Wald-Wolfowitz test. None of them offered really efficient results.", our method makes heavy use of statistics, especially the ANOVA method.

B. Time Measurement Methods

Since cryptographic operations (especially public key operations) are computationally complex, they are also CPU time demanding. Time measurements of cryptographic operations are difficult to make in practice. However, if the attacker is able to gather time measurements related to a cryptographic operation, it may become possible to her to uncover secrets. Our paper deals with a special case of the second step, how to cover the RSA private key. We assume that the attacker was able to collect enough side-channel information, as presented in Section III-A.

As a generic example, assume that the attacker can reach that a computer encrypt arbitrary blocks of plaintext she chooses. If she can do it remotely, she can measure the time between the message being sent out and the message with the result arriving to her. This includes the network round trip time that is a noise in our case. If she can log in to the computer, and run the cryptographic operation locally, round trip times are not added, but the actual CPU load of the computer might still add some noise to the measurements.

C. RSA Implementation Using the Montgomery Algorithm

The RSA public key algorithm [10] makes intensive use of modulo arithmetic. An efficient way of performing modulo arithmetic is the Montgomery Algorithm [6]. Prince's paper [7] is a very good in-detail explanation of it.

We implemented the RSA encryption operation illustrated in Equation 1 using Algorithm 1. The private key consists of the pair(d, N). The attacker knows the modulus (N) that is supposed to be the same constant value for every encryption. The attacker's goal is to recover d. P and C are the plaintext and ciphertext messages respectively.

$$C = P^d (\text{mod N}) \tag{1}$$

Algorithm 1 RSA using modulo multiplication.

1: $C := 1$
2: **for** $i = 1$ to length of d **do**
3: **if** d_i is 1 **then**
4: $C := CP \bmod N$
5: **end if**
6: $P := PP \bmod N$
7: **end for**

Algorithm 1 applies the Montgomery multiplication for the modulo multiplication (in step 4) and square operations (in step 6). The time of a Montgomery multiplication is constant, except when the intermediate result of the multiplication is greater than the modulus. In that case an additional subtraction has to be performed. We refer to this step in the paper as an *extra-reduction step*. For the illustration of the extra-reduction step, Algorithm 2 presents the pseudocode of the Montgomery Algorithm that is used internally by the Montgomery multiplications.

Algorithm 2 Montgomery Algorithm (MM):
$Z \equiv \text{MM}(X, Y) \equiv XYr^{-1} (\text{mod} N)$

1: $Z := 0$
2: **for** $i = 0$ to $n - 1$ **do**
3: **if** X_i is 1 **then**
4: $Z = Z + Y$
5: **end if**
6: **if** Z is an odd number **then**
7: $Z = Z + N$
8: **end if**
9: Shift right the binary form of Z with one position
10: **end for**
11: **if** $Z \geq N$ **then**
12: $Z = Z - N$ // This is the extra-reduction step.
13: **end if**

Considering RSA algorithm (see Algorithm 1), when the i^{th} bit of d is 0 ($d_i = 0$), no multiplication, thus no extra-reduction step is made in the i^{th} round of the for loop. On the other hand, 0, 1 or 2 extra-reduction steps are performed when $d_i = 1$.

Building a mathematical model that models the behavior of the number of extra-reduction steps in a given round for a given key and message is a great challenge. Timing attacks that are based on the estimation of the number of extra-reduction steps executed by the Montgomery Algorithm need to run the algorithm in order to get the number of extra-reduction steps at round i of the looping algorithm, for a given message and key.

D. Statistical Tools Used for Key Recovery

This section presents the statistical tools that we use in our attack scheme, i.e. the one-way analysis of variance (ANOVA) method and the two-sample unpooled t-test. We describe the aim and the main characteristics of each of these methods. We must note that the output of these statistical tools is not interpreted in the traditional way in our attack scheme. We use these methods "only" to generate input values for the decision making on the key bits during key recovery.

1) Calculation of ANOVA F values: One of the statistical methods that we apply in parts is the one-way analysis of variance (ANOVA) [11, 12] method. The one-way ANOVA method tests if the mean of two or more random variables having normal distribution are equal. Given the set of means, the ANOVA will calculate the probability that the observed differences among them could have arisen by chance due to sampling variation.

The null hypothesis states that there are no differences between the means of the different groups, suggesting that the intra-group variance should be equal to the inter-group variance. The alternative hypothesis is that any of the group means differ form the others. The ANOVA method mainly compares a corrected value of the Sum of Squared deviations between groups (SS_{betw}, depicted in Equation 2) and the Sum of Squared deviations within groups (SS_{within}, shown in Equation 3).

$$SS_{\text{betw}} = \sum_{j=1}^{M} n_j (\bar{y_j} - \bar{y})^2 \tag{2}$$

$$SS_{\text{within}} = \sum_{j=1}^{M} \sum_{i=1}^{n_j} (y_{ij} - \bar{y_j})^2 \tag{3}$$

M is the number of groups, n_j is the number of samples in group j, $\bar{y_j}$ is the mean of the sample values in group j, $\bar{y}$ is the overall mean of all sample values, and $\bar{y_{ij}}$ is the value of sample i in group j.

The test function of ANOVA is the F function presented in Equation 4 that has Fischer (F) distribution if the null hypothesis is true.

$$F = \frac{SS_{betw}/(M-1)}{SS_{within}/(n-M)} \tag{4}$$

n is the number of all samples. If the F value is greater then a critical value $F_{1-\alpha}(M-1, n-M)$ then the alternative hypothesis is accepted, i.e., the test concludes that there is a significant difference between the means of the groups. α is the one-sided confidence level for the F distribution. It determines the critical value below which we accept F values as part of the F distribution. The smaller α is, the more powerful the alternative hypothesis is.

In our key recovery algorithm, ANOVA is used to calculate F values. We do not follow the standard decision making steps of ANOVA. We often get high F values (i.e, higher than any reasonable F critical), however, we accept the null hypothesis in our decision making algorithm. This is due to the fact that F can also be much higher if our null hypothesis is not true.

2) Two-sample unpooled t-test: The two-sample unpooled t-test [13, 14] is another statistical method that our attack scheme applies in parts. This test aims to analyze the means of the samples from two independent populations. The null hypothesis is that the means are equals, while the alternative hypothesis is that they are different. The t test function is given in Equation 5. It has Student-t distribution, when the null hypothesis is true, and certain assumptions hold.

$$t = \frac{(\bar{x_1} - \bar{x_2}) - (\mu_1 - \mu_2)}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{5}$$

$\bar{x_1}$ and $\bar{x_2}$ are the estimated means of the two samples; μ_1 and μ_2 are the population means (that are unknown, but supposed to be equal in the null hypothesis); furthermore, s_1 and s_2 are the estimated standard deviations of the samples. Regarding the assumptions of the test, the two populations must have normal distributions or more than 40 samples for $(n_1 + n_2)$, and the standard deviation of the populations can be unknown and unequal.

Given a confidence level α, it is possible to calculate a two-sided confidence interval with an upper and a lower bound value using the Student-t distribution function. If the test function gives a value within the confidence interval, the null hypothesis is accepted. The mean values are considered significantly different, if the test function results a value outside the confidence interval.

In our attack scheme, we skip the decision making part of t-test. t function provides us a long-term objective value that characterizes the degree of separation of the two samples.

III. Timing Based Key Recovery

In this part we describe the attacker model, and the concept of our attack.

A. Attacker Model

In our proof-of-concept attack, we made simplifications regarding the information known to the attacker. We suppose that the attacker knows the total number of extra-reduction steps (T_{total}) for a number of messages P. The attacker uses this knowledge to guess the key d (see Equation 1) bit by bit, from $i = 1..n$ in off-line mode.

Several papers [1, 3] suggest that the number of extra-reduction steps correlates with the total coding time, even though we did not analyze this dependency. If the attacker is able to measure coding times, the attack scheme would

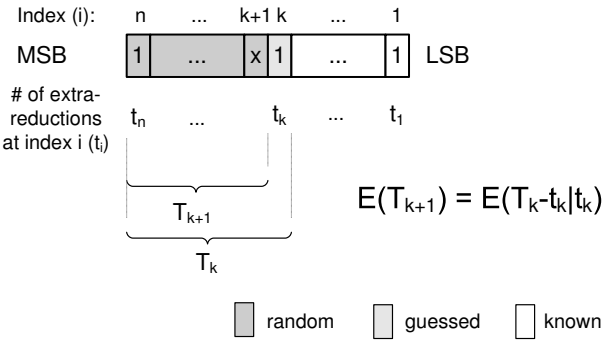

Fig. 1. The concept of our attack.

also work using that information, since correlation with the total number of extra-reductions is supposed. The task of the attacker would in that case be more difficult, since measurements always introduce noise. However, by using more samples and by applying noise filter methods [15], the same attack scheme works. Our starting point is, however, that the attacker has accurate knowledge of the total number of extra-reduction steps for each message P, because our aim now is to concentrate on the development of the key recovery algorithm.

B. The Concept of the Attack

The concept of our key recovery method is presented in this section. The key recovery is based on the fact that Algorithm 1 utilizes key d, bit by bit, from its Least Significant Bit (LSB) to its Most Significant Bit (MSB). These two bits are always one, since MSB must be one, otherwise it would not be the MSB, and LSB must be one since d is an odd number.

The idea of key recovery is depicted in Figure 1.

We make the assumption, that the attacker already knows the correct key prefix from the LSB to key bit position $k-1$, i.e. d_i is "known" from $i = 1..(k-1)$. This also implies that she can calculate the number of extra-reduction steps t_i for $i = 1..(k-1)$ made by Algorithm 1 for any message P. At this phase, she wants to guess bit d_i for $i = k$. The remaining part of key d is unknown. We make the assumption that the unknown bits of the key cause independent and random number of extra-reduction steps, or more generally, the remaining time is a random value with finite mean and variation. The attacker does not know the exact number of extra-reduction steps for these key bits. However, she can calculate T_k, the remaining number of extra-reduction steps for each message, knowing the total number of extra-reduction steps.

The attacker guesses the key bit d_k. The assumption in our idea is, that if the guess is correct, the calculation of t_k the number of extra-reduction steps for bit k can be done correctly for each message P. Since one more key bit becomes known, the length of the random part of the key becomes one bit less, and the attacker has more accurate information to calculate the remaining time in the next round. In our proof-of-concept attack, the attacker calculates the remaining number of extra-reduction steps (T_{k+1}) for each message using Equation 6.

$$T_{k+1} = T_k - t_k \tag{6}$$

Our assumption is that the remaining times (T_{k+1}) are independent random values that come from one population with the estimated mean $E(T_{k+1})$. However, we say that this assumption is true only if we calculated with the correct t_k for the messages. If the key bit d_k was guessed incorrectly, then we would apply an incorrect (random) value in the calculation of T_{k+1} values. Thus, we would get other T_{k+1} values. Note, that these assumptions may not only hold for the number of extra-reductions.

We classify the T_{k+1} values into separate groups based on the values t_k. If t_k is correct, then the means of the separate samples must be equal, however, in case of random t_k values this assumption does not hold.

In order to decide, whether the means of separate groups of T_{k+1} are equal or not, we apply the F test function of the ANOVA method (see Section II-D1). The null hypothesis of ANOVA method is given with Equation 7.

$$H_0 : \mu = E(T_k - t_k|t_k), \text{for all } t_k \tag{7}$$

We have only three groups, because t_k can have three values. We suppose the value 1 in our key bit guesses, because 0 produces only one group where $t_k = 0$. We expect, that we get low F values in case of correct guesses, and high F values for incorrect guesses of d_k. However, in order to apply F values for the decision on the correctness of guesses, we need to analyze, what low and high F values actually mean. This is presented in Section III-C in detail.

We base our key recovery algorithm on the previously described idea. We also rely on the assumption that the more key bits become known, the fewer unknown factors influence the remaining times, thus the more accurate values we are able to give as input to ANOVA, hence F values should become more separated.

C. Analysis of F values with Measurements

The values generated by the F test function of the ANOVA method need to be analyzed before using them in the decision making on the correctness of the guess. Normally, we expect $F < F_{1-\alpha}(M-1, n-M)$ when the key prefix (d_i for $i = 1..(k-1)$) and the guess ($d_k = 1$) is correct. However, we have experienced that even in this case F values might be higher then the critical value. This led us to reject the idea of using the critical value of F for decisions. On the other hand, we have seen that we get even higher F values if the guess is incorrect, i.e., $d_k = 0$. Hence, we concluded that even these biased F values could be used as input for decision making on the correctness of guesses.

The following Sections describe the main behavior of F values for three different cases. In the first case, we apply the correct key prefix in the calculation of F values. In the second case, we suppose some incorrect bits in the key prefix, and analyze the trend of F values in that case. In the third

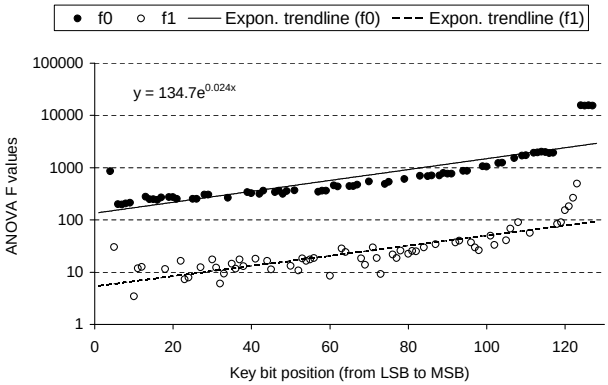

Fig. 2. F values in case of correct key bit guesses using 100000 samples.

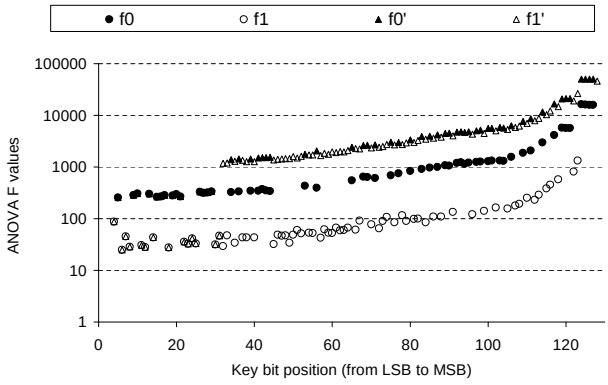

Fig. 3. F values in case of incorrect key bit guess at key bit position 30 using 100000 samples.

case, we analyze the effect of the number of samples on the F values.

1) Making Correct Guesses on Key Bits: Figure 2 shows the F values calculated with correct key prefixes. When we guess the key bit d_k, we make the assumption that it is 1, and we calculate the value F using the correct key prefix, i.e. the correct d_i bit values for $i = 1..k-1$. Looking the F values related to $f1$, we can clearly see, that if the key bit d_k is really 1, then, the F tests produce low values. However, if d_k was in reality 0, then they produce high values, as shown by the $f0$ values. The meaning of low and high is relative, regarding the logarithmic scale of the axis of F values. An F value considered to be high at a lower bit position might be regarded as low on a higher bit position. This behavior of F values complicate the use of F test in decision making on the correctness of guessings.

2) Making Incorrect Guesses on Key Bits: Figure 3 illustrates the behavior of F values when an incorrect key bit is used at key bit position 30. $f1$ and $f0$ represent the F values in case of correct guesses when the correct key bit value is 1 and 0 respectively, while $f1'$ and $f0'$ show the F values of the same bits, when the bit in position 30 is guessed incorrectly. It can be seen that F values give us very soon feedback on the incorrect guess.

We experienced that the F values become high and stay on that level one or some bits after the first incorrect guess. The values of subsequent key bits and the correctness of subsequent guesses do not influence the subsequent F values. F values remain on that high level. This fact enables the use of F values in key recovery.

3) Dependence on the Number of Samples: Figure 4 illustrates the behavior of F values in case of correct key prefix bits used in the guesses. However, now the number of samples is low. In the case depicted by the figure, the attacker has 5000 samples for the ANOVA calculation.

The higher the number of samples is, the better separation between the F values of key bit values one and zero can be reached. In Figure 2 and 3 the clear separation is due to 100000 samples.

4) Challenges in Using F values for Key Recovery: Challenges in using F values for key recovery can be clearly seen

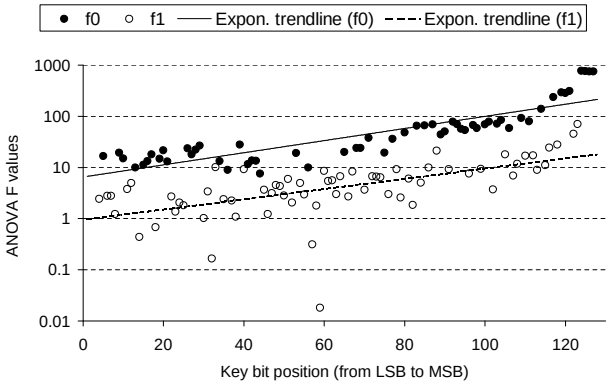

Fig. 4. F values in case of correct key bit guesses using 5000 samples.

from the previous illustrations and our measurements. We analyzed F values using different keys, different number of samples, and by introducing incorrect key bits in one or more positions. We concluded that the main behavior of F values is similar. However, the F values are relative. They show a high variance in the interval of 10^1 and 10^5. The value of F depends on the position of the guessed key bit, the value of the key d, the number of samples. The main challenge, thus, is the absence of an absolute F critical value.

IV. The Novel Algorithm

Most timing attack algorithms guess the key bit by bit. The advantage of this technique is that it is fast, the disadvantage is that if one single bit is missed, it will not find the correct key. Our algorithm addresses this problem.

A naive approach would be to start building the key, and at every point decide whether the prefix we have so far is correct or not. If it is correct, we can go on, and guess the next bit, if it is incorrect, we should "backtrack", and try a different branch. Unfortunately this does not solve the original problem. If we make the wrong decision and backtrack at a single point, we will never find the correct key any more.

To summarize up our approach: We use a more "fuzzy" method. We guess one bit at a time, but instead of keeping track of only one key prefix and progressing linearly, we store

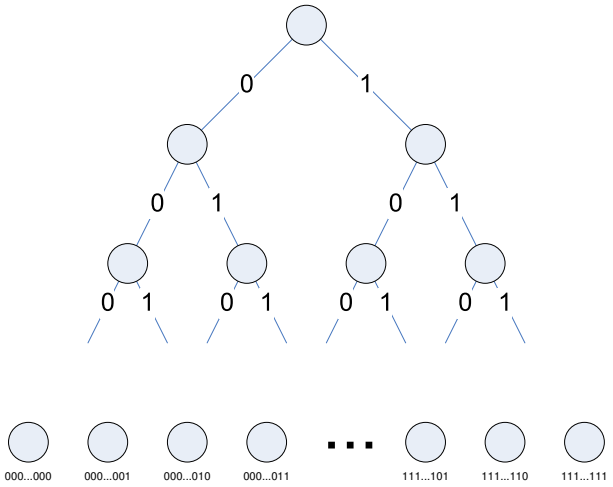

Fig. 5. Key tree: a full binary tree of depth n.

several key-prefix candidates all at the same time, and always extend the most probable one.

The following Subsections introduce our algorithm in details.

A. Key tree

According to Kerckhoffs' principle, all the details of the algorithm are known to the attacker except for one secret key. The attacker tries to reconstruct this key. If the length of the key is n bits, there are 2^n possible keys. Consider a binary tree of depth n. We will call this the *key tree*, see Figure 5. The leaves of the tree (at the bottom of Figure 5) represent the possible keys. The other nodes represent the key prefixes. At every level there is a 0 (left) and a 1 (right) branch. We will call a node a *zero-node* if it represents a key bit 0 (thus it is the left child of its parent) and we will call it a *one-node* if it is the right child of its parent. Every node in the tree is either a zero-node or a one-node, except for the root node (which represents the empty prefix of the correct key).

Guessing the key one bit at a time, from the beginning to the end, would seem finding a path from the root (top in Figure 5) to the correct leaf. There is only one correct key[1], thus, there is only one correct path. If the attacker misses one single bit, she has no chance to find the correct key.

Our algorithm is more advanced than this. We start visiting the nodes of the key tree from the root. Our goal is to find (visit) the leaf node corresponding to the correct key (thus recovering the key). The goal is to visit as few nodes as possible during the process. The theoretical minimum is the depth of the key tree (key length, n), the worst case is visiting all the nodes ($2^{n+1}-1$), which is almost two times as many steps as an exhaustive search among all possible keys (2^n). We have to find a balance between trying not to miss the correct path and advancing as deep down as it is possible.

[1]Considering some algorithms it is possible that there are more than one correct keys. However, there still has to be very few keys, unless guessing would become feasible.

Our achievement is a data structure for storing all visited nodes (key prefixes) efficiently and an advanced way to select the next node to visit.

B. Goodness of Nodes

We introduce a measure called *goodness* for each node visited. This goodness represents how *good* we consider that key prefix, or how *probable* we consider it to be correct. As all the prefixes of the correct key are correctly guessed prefixes, we just have to follow the path of nodes with high goodness value to find our way down to the node representing the correct key. *Goodness* should be defined in a way so that it is as high as possible for correctly guessed prefixes, while as low as possible for incorrectly guessed prefixes.

1) Local Goodness of Nodes: Each node (prefix) has a local goodness value assigned to it. Local goodness is based on the F values of ANOVA, see Section III-C.

There are two things that we assume. One is that the F value of a one-node is close to the F value of the one-nodes above it, and an F value of a zero-node is close to other zero-nodes' F values above it. The second assumption is that the F values of one-nodes and zero-nodes are significantly different from each other on the path from the root to the correct key, and are close to each other elsewhere.

It is easy to see that these assumptions hold, see Section III-C. The problem here is that we have terms like "close to each other" and "close to other values". This have to be formalized to be used in an algorithm, and this is exactly what local goodness values are. They describe how close these values are to each other.

We defined the local goodness as the product of two factors. One expresses how much the F values of zero-nodes and one-nodes are different, and used the t-test for it (see Section II-D2). The other factor expresses how close the F value of a node is to its group. A one-node's F value should be close to that of other one-nodes, a zero-node's F value should be close to other zero-nodes' F values. For the calculation of F we used ANOVA, see Section II-D1.

We have also normalized the local goodness (i.e. scaled it to be a value between 0 and 1), a goodness of 1 means that the node is on the path to the correct key (i.e. it's a correctly guessed prefix) with a high probability.

2) Cumulated Goodness of Nodes: Goodness values are probabilistic, so it might turn out that a node that is not on the correct path has a higher goodness value by chance than another node that is on the correct path. This happening has higher probability if there are fewer samples. We want our algorithm to be robust and make sure that these kind of "false" goodness values do not misdirect the flow of the algorithm, and make it lose the correct path.

For this reason, when computing how good a node is, we wanted to consider the goodness of other nodes above that node. The idea came from the fact that if a node is on the correct path (represents a prefix of the correct key), all its ancestor nodes also represent (shorter) prefixes of the correct key. This led to the introduction of *cumulated goodness*.

Cumulated goodness is a weighted sum of the local goodness of the node and the local goodnesses of all the ancestor nodes of it. The further away an ancestor is, the smaller weight is assigned to it. Let x denote the local goodness of the node, $x^{(1)}$ is the local goodness of its parent, $x^{(2)}$ is the local goodness of its parent's parent, etc. We defined one parameter: λ, its value is between 0 and 1, and it describes how fast do we "forget". Equation 8 shows how the cumulated goodness (C) of a node is computed.

$$C = \lambda x + (1-\lambda)\lambda x^{(1)} + (1-\lambda)\lambda^2 x^{(2)} + ... + (1-\lambda)\lambda^n x^{(n)} \quad (8)$$

The cumulated goodness of a node (C) can be easily computed from its local goodness (x) and the cumulated goodness of its parent ($C^{(1)}$), see Equation 9.

$$C = \lambda x + (1-\lambda)C^{(1)} \quad (9)$$

The higher λ is, the less we take the goodness of the node's ancestors into consideration when computing its cumulated goodness. Values between $0.5...0.6$ seemed to work the best.

C. Next Node Selection

As it was described earlier, our algorithm start from the root node of the key tree and tries to find a path down to the leaf corresponding to the correct key. At every given time, there are some nodes that have already been visited, and there are some that have not been. Our algorithm always keeps going from top down. It means that when it decides which node to visit next, it only selects from the children of already visited nodes in the key tree. A visited node with at least one unvisited child is a node where our algorithm can continue from. These nodes are called possible *continuation* nodes.

Algorithm 3 Our advanced timing attack.

```
1:  L :={ root of the key tree }
2:  repeat
3:    n := node from L with highest goodness
4:    Remove n from L
5:    for all  x as child of n do
6:      if x is the correct key then
7:        Correct key found
8:      end if
9:      Compute the goodness of x
10:     Add x to L
11:   end for
12: until Correct key is found.
```

Algorithm 3 shows the structure of our attack algorithm. L denotes a set of continuation nodes. We start with only one node in the set: the root of the key tree (step 1). Note that an arbitrary goodness value can be assigned to the root node. However, this value should not be very high due to its influence in the cumulative goodness computation in the nodes near the root node. At every step we select the node with the highest cumulated goodness (see Section IV-B2) from L, and continue from there. Let n denote this selected node (step 3). We remove this node from L and add its two children instead (steps 4 and 10). We continue this until the correct key is found.

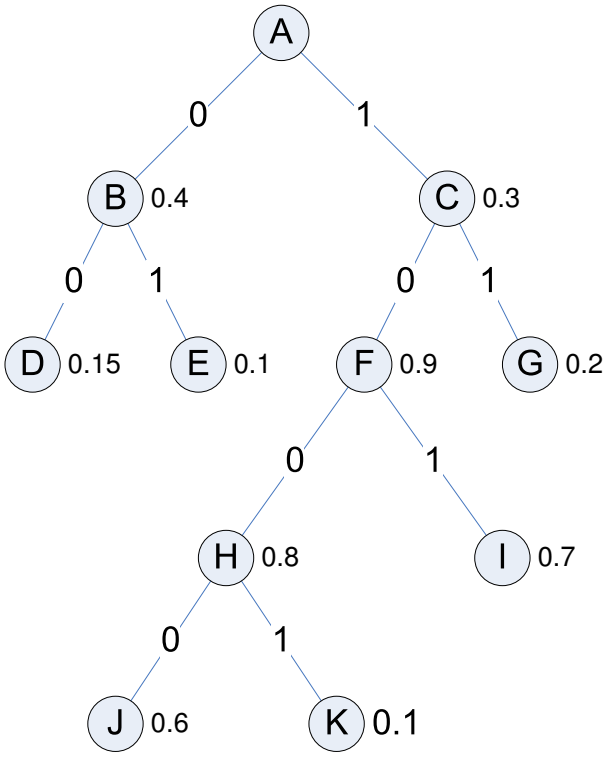

Fig. 6. An example path on the key tree with cumulated goodness values.

Note, that the number of nodes in L is always increasing. Figure 6 shows an example. The number on the right side of each node shows the cumulated goodness of the node. The algorithm started from the root node (A). The root node does not have a goodness value, because it is in the list of continuation nodes only at the very beginning of the algorithm. When it is the only node in the set of continuation nodes, the algorithm will select it anyway, while looking for the node with the highest goodness. In the first step nodes B and C were added with the cumulated goodness values of 0.4 and 0.3 respectively. After this step, the list of possible continuation nodes L contains B and C.

Then, Node B is chosen to continue from, because it has the highest goodness value. This results that Node B is removed from L, and D and E are added with cumulative goodness values of 0.15 and 0.1 respectively. At this point the set of continuation nodes is $L = \{C, D, E\}$.

In the next step, C is chosen to continue from. Note, that this is an example where algorithm "backtracks". B seems more promising than C, so we try to continue from B. But later, it turns out, that C is still more promising than all the possibilities that could be found under B. So at this point we switch to C. It is important to see, that later we still might switch back to D or E. So, as the algorithm continues from C, it exposes nodes F and G. At this point $L = \{D, E, F, G\}$.

Then the algorithm continues from F (exposing H and I), then from H (exposing J and K). Figure 6 shows this state. Now $L = \{D, E, J, K, I, G\}$. At this point the algorithm will do another "backtrack" and switch to I, because it has the highest goodness value assigned to it (of all the nodes in L).

Note, that if we assign the cumulated goodness of a node to the incoming link, the problem could be interpreted as finding the longest path in the tree. Algorithm 3 can be seen as a solution for the longest path problem. However, in this tree the goodness values are not known in advance. The innovative part of our work resides in the way how we assign goodness values to the nodes (or links) using statistical tools, that enables to find the correct key with a high proportion of correct key-bit guesses.

D. Refinements

We have presented in this section our algorithm in its simplest form. There are several possible refinements and extensions. Here, we present the most important ones.

1) Initialization of the Algorithm: During testing our algorithm in practice (see Section V), we have realized that our algorithm performs weakly at the very beginning of the key. The correlation between the ANOVA F values and the correctness of the key prefix is not strong enough for our algorithm to work efficiently. To avoid the exploration of large subtrees of the key tree we have designed and implemented a special initialization of the algorithm. The idea is that instead of starting at the very first bit, all prefixes of a given key length are precomputed. Let p denote this key length. There are 2^p key prefixes of length p which has to be precomputed, and all of them are put into the list of possible continuations (L). We have chosen $p = 10$ to find a balance between feasibility and efficiency. Our algorithm starts traversing down from 1024 nodes. This is the reason why the key prefixes don't start from 0 in Figure 7.

2) Switching to Brute Force: Another phenomenon we have observed is that the correlation between the ANOVA F values and the correctness also decreases sharply at the very end of the process, i.e., when there are only a few bits left. This usually happens at the last 8-10 bits, and it's independent of the actual key length.

A straightforward improvement, that we have not implemented yet, would be to use a brute force approach, once we get close enough to the full key length. It might be difficult to tell when exactly we should switch to brute force. However, it is important to note, that even though our ANOVA F value based algorithm becomes less efficient at the last few bits, it only happens that close to the key length that even a brute force solution is feasible from that point on. Switching to brute force when there are only 10 bits left would be reasonable.

V. Results

The following section presents the characteristics of our key recovery algorithm. We demonstrate its behavior during the breaking of a 128-bit key. The section discusses the efficiency of the algorithm, finally, mentions further development possibilities.

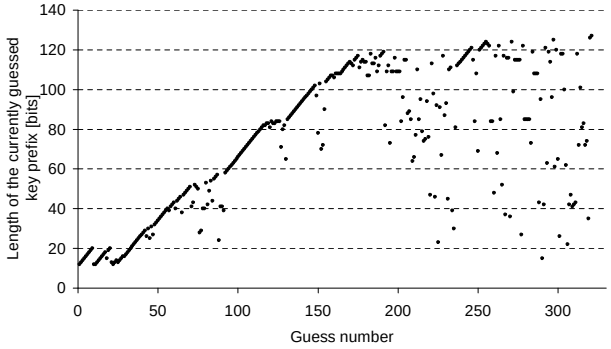

Fig. 7. Length of the key prefixes of the selected key tree nodes during the key recovery steps.

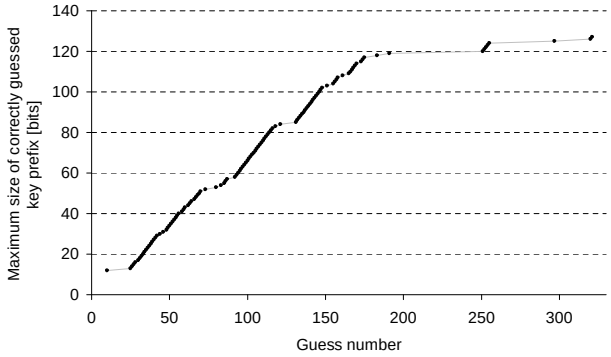

Fig. 8. The size of the last longest correctly guessed key prefix during the steps of the attack algorithm.

A. Recovery of a 128-bit Key

In the following figures, we depict the main characteristics of the node selection procedure of our key recovery algorithm. The algorithm is searching for a 128-bit key, based on 10000 samples. The correct key, unknown by the algorithm, is $d =$ 6a5c43bce04b9f8a9171e23a8fbb4fd1 using hexadecimal format[2].

Figure 7 shows the length of the key prefixes belonging to the key tree nodes, that the algorithm selects during the node selection steps, while it searches for the correct key. As it was explained earlier, the minimum key prefix length is 10 bits, because all the 1024 prefixes of length 10 have been precomputed, and the attack started from there. Also note, how the efficiency gets worse as the full key length is approached.

Figure 8 presents the longest key prefix that has been guessed correctly as a function of the node selection steps.

Figure 9 presents the hamming distance between the key prefix of the parent of the selected node and the correct key prefix of the same length observed during the key recovery steps.

The main conclusion that can be read from Figures 7, 8 and 9, is that the algorithm follows sufficiently well the correct path in the key tree, until the final eight bits to recover, where it gets lost for a while. However, the algorithm finds the good

[2]In fact, this key has been generated for 128-bit RSA, and the exact length of the key is 127 bits.

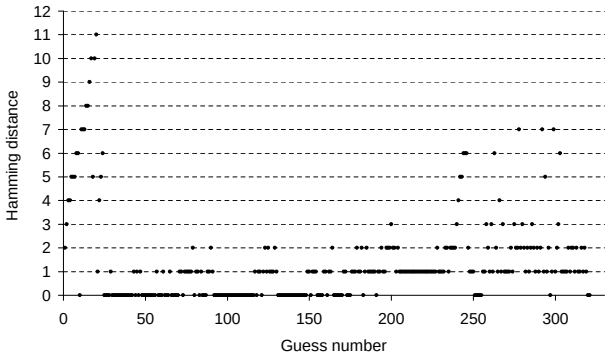

Fig. 9. Hamming distances – observed during the key recovery – between the correct key prefix and the one that is known (and supposed to to be good) by the actually selected nodes.

path relatively soon.

In this specific case, the algorithm has made 320 guesses (as opposed to the optimum of 118, because 10 bits were precomputed). It has reached the 119 bit prefix of the correct key in the 191^{th} step. In other words, 93.7% of the key has already been recovered but then rejected at 59.6% of the key recovery process. Switching to a brute force approach might help this, see Section IV-D2.

B. Further Development Possibilities

We can see that the algorithm has difficulties to find the last key bits, and it steps back to much shorter prefixes. As a possible solution, the algorithm could be extended with a brute force key search mechanisms which can be used for the last few bits.

Another problem, we have experienced with the algorithm was the effect of long-runs of bits 0 or 1 in the key d. In this case, it can happen that the algorithm does not consider enough F values on the actual path from the actual node, thus, it does not get enough F values for the opposite bits. This results in the situation when the two-sample unpooled t-test function does not get enough (or any) samples from one of the populations. A possible solution for this problem is to set a minimum and a maximum value for the look-back distance, i.e., the number of F values to consider. Typically we apply the minimum value for the included F values, but we have the possibility to extend the included number of F values. The minimum and maximum values would in this case depend on the key length, since the distribution of the lengths of long-runs also depends on the key size.

VI. Concluding Remarks

We have presented an advanced timing attack scheme on cryptographic algorithms. The attacker can use our algorithm to break a cryptographic algorithm by reconstructing the secret key. As a proof-of-concept, the key recovery method has been demonstrated on a specific implementation of the RSA algorithm efficiently revealing the 128-bit secret key.

We have shown the complete key recovery in practice using side-channel information on the total number of extra-reduction steps of the Montgomery Algorithm utilized by the RSA implementation.

We have introduced a new concept on key trees and goodness values assigned to key prefixes, which makes the recovery algorithm examine only a very limited part of the key space, even if the decision criteria for guessing the key bits are highly biased. This concept can be extended to any other timing attack.

In the proof-of-concept example, the key recovery algorithm has found the correct 128 bit key using 320 guesses. Moreover, 93.7% of the key bits have been already found after 119 guesses. In the future, we plan to make refinements for the algorithm.

We have also shown some refinements to the original algorithm, and proposed a way how its efficiency may be further improved at the last few bits of the key.

References

[1] Paul C. Kocher, “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems,” in *CRYPTO 1996*, 1996, pp. 104–113.

[2] W. H. Wong, “Timing attacks on RSA: revealing your secrets through the fourth dimension,” *Crossroads*, vol. 11, no. 3, pp. 5–5, 2005.

[3] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, “A Practical Implementation of the Timing Attack,” in *CARDIS ’98: Proceedings of the The International Conference on Smart Card Research and Applications*. London, UK: Springer-Verlag, 2000, pp. 167–182.

[4] D. J. Bernstein, “Cache-timing attacks on AES,” 2004. [Online]. Available: http://cr.yp.to/papers.html#cachetiming

[5] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 26, no. 1, pp. 96–99, 1983.

[6] Peter L. Montgomery, “Modular Multiplication Without Trial Division,” *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr 1985.

[7] B. J. Prince, “Scalable Montgomery Multiplication Algorithm,” Oregon State University, Department of Electrical & Computer Engineering, 2002.

[8] A. Kerckhoffs, “”la cryptographie militaire”,” *Journal des sciences militaires*, vol. IX, pp. 161–191, Feb 1883.

[9] Helena Handschuh and Howard M. Heys, “A Timing Attack on RC5,” in *Selected Areas in Cryptography*, 1998, pp. 306–318.

[10] B. Kaliski and J. Staddon, “PKCS #1: RSA Cryptography Specifications Version 2.0,” RFC 2437 (Informational), Oct. 1998, obsoleted by RFC 3447. [Online]. Available: http://www.ietf.org/rfc/rfc2437.txt

[11] O. J. Dunn and V. A. Clark, *Applied statistics: analysis of variance and regression (2nd ed.)*. New York, NY, USA: John Wiley & Sons, Inc., 1986.

[12] Sahai, Hardeo, Ageel, and Mohammed I., *Analysis of Variance, Fixed, Random and Mixed Models*. Birkhäuser, 2000.

[13] J. M. Utts and R. F. Heckard, *Statistical Ideas and Methods*. South Melbourne, Victoria 3205, 80 Dorcas Street, Australia: Cengage Learning Australia, 2006.

[14] R. J. Larsen and M. L. Marx, *Introduction to Mathematical Statistics and Its Applications*. Prentice Hall, 2006.

[15] S. W. Smith, *The scientist and engineer’s guide to digital signal processing*. San Diego, CA, USA: California Technical Publishing, 1997.