622.755 – Crypto Engineering Winter Term, 2020/21

Handout 1: Introduction to Side Channel and Fault Attacks: RSA Case Study

Elisabeth Oswald

1 General Comments

This handout supports the delivery of the Crypto Engineering unit. For each week, there are slides, some videos and sometimes handouts. I suggest if there is a handout to read the handout first. It will give you some background information and explain important concepts, give examples, and suggest some exercises as well as hints for further reading. Anything under further reading will enhance your understanding, but is not strictly required to be able to get on with the coursework.

The videos are designed to talk you through the notes as well as the exercises and make explicit links to the coursework. I will often use analogies in videos to explain things, whereas the notes, whilst explaining the same ideas, will be more formal.

For the first two weeks you need to have a general understanding of basic cryptography (in particular RSA) such as provided in Cryptography (Summer Term).

This handout covers all introductory aspects of this course. There are seven associated videos available from YouTube via an unlisted playlist:
`https://www.youtube.com/playlist?list=PLQ5hY1xU49CBkXsaK5TgUiQsy4jAZ7-xp`.
The videos are numbered, and you are looking out for numbers 6–12.

2 What you should get out of this.

This initial part on “breaking crypto” gives you a first glimpse of implementation attacks. After engaging with this content you should be able to:

- name a few types of sources for information leakage,
- name a number of side channel attacks,
- explain the difference between active and passive adversaries in the context of implementation attacks,
- utilise timing information to recover secret data,
- sketch how a differential side channel attack works,

- sketch how a fault attacker typically proceeds,
- have an awareness of the process by which a cryptographic algorithm/protocol is translated into a concrete implementation and pinpoint where potentially theoretical assumptions cannot be realised in practice.

3 From theory to practice

Cryptography, at its' core, has been always been a discipline that connects theory with practice. After all, it is "easy" to construct an information theoretically secure (thus "unbreakable") encryption scheme, e.g. the Vernam cipher. You may recall that the one-time pad simply exclusive-ors plaintext and key (we assume that there is some suitable binary representation of them). For the one-time pad we can prove information theoretic security assuming one chooses the key uniformly at random, at the same length as the plaintext. Which in theory sounds easy, but in practice is not because not only could keys be very long (as they have to match the plaintext length), you also have to transmit this key information securely (i.e. it needs to stay secret and you need to authenticate the recipient). If you could transmit the key easily in practice, you could as well transmit the original message over such a "magic" channel ...

Thus in practice, we need to make compromises: e.g. rather than generating truly random key material freshly for each message, we derive randomly looking key material, and we reuse it (for several blocks of a message, i.e. we use block ciphers). We also have to find a way to exchange keys over public channels (in the absence of "magic" channels) using public key cryptography. Finally, nobody would expect end users to engage with any of these tasks, thus we need to translate the mathematical description of a cryptographic scheme into some executable code.

Implementation attacks take advantage of the fact that when we translate a mathematical description of a cryptographic scheme into something that is executable on a machine, we cannot bring forward all of the assumptions made in the proof(s) that may accompany the original description of the scheme. Let's look at this at the concrete example of RSA encryption.

3.1 RSA revisited

The textbook RSA scheme works as follows. Given a key pair consisting of a public key pk and a secret key sk, with $pk = (e, N = pq)$ and $sk = d = e^{-1} \mod \phi(N)$, the encryption of a plaintext m is given as $c = m^e \mod N$. Decryption is given as $m = c^d \mod (N)$.

RSA in its "textbook" form gives weak theoretical security guarantees because of its homomorphic property, and thus should be used only when the input is sufficiently protected. In practice, when used for encryption, this is achieved by using OAEP. This padding function requires randomness r as well as some functions G and H that are supposed to behave like random functions and respectively expand and compress their input.

The idea of a "random function" is that it resembles a "random looking string". Unfortunately we don't have real-world constructions discovered that are truly "random functions" (I put everything in inverted commas because I don't intend to formally define any of these concepts and rather appeal to intuition). What cryptographers tend to use as "substitutes" are hash functions, e.g. SHA-1 (in the original OAEP paper), SHA-3, etc.

The message m that enters the RSA (textbook) encryption function given an input x is defined as

$$m = (x||0^{k_1} \oplus G(r)||r \oplus H(x||0^{k_1} \oplus G(r))). \tag{1}$$

(Thus one appends k_1 zero bits to the message x, samples a value r (of k_1 bits) from random, and exclusive-ors $\oplus$ the respective elements in a Feistel-like manner to create a "ciphertext" aka the encapsulation m of the original message x. The specific choice of G and H determine how much "space" there is for the message and zero padding.)

Thus RSA encryption is applied to m (rather than x) and results in a ciphertext $c = m^e \pmod N$.

Decryption of c works by first applying RSA decryption to c in order to recover m. Then, secondly by splitting m into its' two parts $(x||0^{k_1} \oplus G(r)$ and $r \oplus H(x||0^{k_1} \oplus G(r))$, and "decrypting" this Feistel cipher to recover x (i.e. by applying H to $(x||0^{k_1} \oplus G(r)$ and exclusive-oring this with $r \oplus H(x||0^{k_1} \oplus G(r))$ one recovers r, and by applying G to r one can recover $(x||0^{k_1}$ and thereby x).

Defining security that is meaningful for the real world use of RSA is not straightforward. Cryptographers' strategy historically has been to try and capture the "intent" and "resources" of a very general and powerful adversary. Thus for any security definition, one spells out exactly what resources an adversary has access to (anything else is out of the scope of a proof), and what they are aiming to achieve. In the case of (public key) encryption, the goal of an adversary is to learn "something" about an encrypted message. This goal can be defined via a game played by the adversary (and supervised by the challenger): the challenger is something like the "master" who sets up the game by sampling a random bit b, which is unknown to the adversary. The challenger also produces a key pair. The secret key sk is embedded it into a decryption oracle $\mathcal{D}$, with which the adversary can interact. The adversary get's the public key. The challenger then asks the adversary to produce two messages m_0 and m_1, from which the challenger selects one (according to the secret bit b). This message m_b is encrypted by the chal-

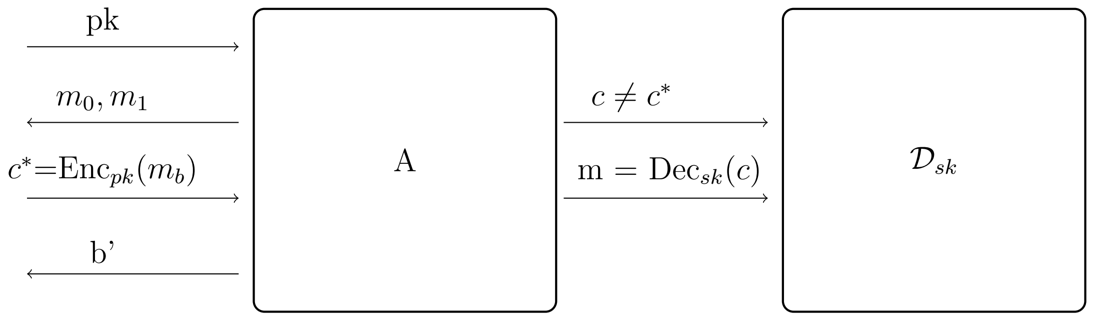

Figure 1: IND-CCA game for PKC.

lenger to produce the challenge ciphertext c^*, which is given back to the adversary, whose challenge is to tell whether m_0 or m_1 has been encrypted. This is repeated many times, and the adversary can select a new message pair each time. I have visualised this game in Fig. 1.

The adversary in this picture can make many calls to the decryption oracle (any number that can be expressed as a polynomial in the security parameter of the scheme, which for simplicity we assume corresponds to the length of the secret key). It is also the adversary who chooses the messages, hence they can format them in a very specific manner if they wish to do so (for as long as they are equal in length). Thus any scheme that meets this definition has to ensure that special properties of input messages have to be somehow concealed, which is exactly what the OAEP transformation achieves. But the adversary gets very little in terms of help otherwise. From the perspective of provable security from decades ago this may have made sense: after all the job of a traditional, mathematical cryptanalyst is to work with plaintext and ciphertext pairs only. However, in real life, adversaries will choose whatever information is available and they may not care much for the specified rules of some theoretical game.

4 Real life attacks on RSA

We now look at a number of attacks, which are all applicable in real world scenarios, that violate the rules of the IND-CCA security game.

4.1 Attacks using Erroneous Ciphertexts

One of the first implementation attacks that was reported for RSA decryption was due to Boneh et al. [2]. It is a classical example of a "fault attack" and their technique can break RSA decryption with two ciphertexts (one reference and a faulty copy). With a simple modification their technique can also be applied to RSA based signature schemes, in which case a single faulty signature suffices [1].

The assumption in a fault attack is that somehow, one can force a device to make a mistake when performing an operation. There are a number of ways in which an adversary can achieve this: devices require power to function, and some even take in an external clock signal. Hence where available, such input signals can be altered to achieve the desired effect. But there are other ways too: by exposing parts of memory and shining light on it, by using EM pulse generators, lasers, focused ion beams, etc. A very readable (but slightly dated) survey of fault attacks is by Giraud and Thiebauld [4].

4.1.1 A Generic Fault Attack

We use an example where the RSA decryption operation is practically realised with the popular binary algorithm (aka 2-ary left to right algorithm, aka square-and-multiply algorithm). This algorithm scans the exponent bits of d, squares the intermediate m in each step but multiplies m by c only if the respective exponent bit is equal to one, see Alg. A. The algorithm thus recovers m from c. To recover the actual message x, a further algorithm that removes the OAEP padding has to be applied to m.

Alg. A: RSA Decryption using the Binary Algorithm

The algorithm computes $c^d \pmod N$ using the left-to-right binary algorithm, and delivers m as return value.

```
input: int N, int d=($d_w, d_{w-1}, \ldots, d_1, d_0$), int c
output: int m
begin
  m = 1
   for $i = w-1$ down to 0
     m $= m \cdot m \pmod N$
      if $d_i == 1$
        m $= m \cdot c \pmod N$
   end
   return m
end
```

A generic fault attack (i.e. technique that can be applied to any cryptographic algorithm) is as follows. We assume that the adversary has the ability to induce faults in specific bits of specific intermediate values that occur during the processing of RSA decryption. In particular we assume that the adversary is able to **set** the value of specific bits d_i. Let us assume that the fault causes d_i to be 0. With this in mind a generic fault attack proceeds as follows. The adversary utilises an encryption oracle to obtain a valid ciphertext c for some arbitrary message m. Then the adversary uses the decryption oracle with c and induces a fault in d_{w-1}, i.e. $d_{w-1} = 0$. The resulting decrypted message

m' can be either unchanged (i.e. $m' = m$, this happens if d_{w-1} was 0 anyway) or $m' \neq m$ (this happens if d_{w-1} had been 1. In the latter case it is likely that the decryption would fail during the OAEP decoding step and output an error. Either way we learn the value of the bit d_{w-1}. By applying the same principle to the remaining bits of the d we can recover d in no more the $w - 1$ steps.

This attack applies even if we utilise RSA with some secure padding, and it does apply equally if no secure padding is used.

4.1.2 A Fault Attack on CRT Decryption

The generic fault attack requires an adversary to induce very precise faults. An attack that exploits properties of CRT decrypted ciphertexts does not have the same requirements. It will work for any arbitrary fault for as long as this fault occurs in only one of the two CRT decryption parts. In addition the adversary needs access to the raw RSA decryption value (i.e. before the OAEP decoding step). The fault will most likely cause the decoding step to fail, and thus if only the result of OAEP decoding was made available to the adversary this would render the attack impractical.

Alg. B: RSA Decryption using the CRT

The algorithm computes $c^d \pmod N$ by taking advantage of the CRT, and delivers m as return value.

```
input: int N, int dP, int dQ, int p, int q, int qInv,
int c
output: int m
begin
  mP = $c^{dP} \pmod p$
  mQ = $c^{dQ} \pmod q$
  h  = $qInv \cdot (mP - mQ) \pmod p$
  m  = $mQ + h \cdot q$
  return m
end
```

The attack is due to Boneh et al. [2] and can often be found called the “Bellcore attack” in the literature owing to the fact that the authors were with a company called Bellcore at the time). As in most fault attacks we assume that the adversary has a reference ciphertext c and it’s corresponding correct decryption m.

Without restricting generality we assume that the adversary induces a fault in the the step when mP is computed. This will produce a faulty message m' after the CRT recombination step: $m' = mQ + q \cdot (qInv \cdot (mP' - mQ) \pmod p))$. Using the difference $m - m'$ enables the recovery of a factor of N using a simple gcd calculation.

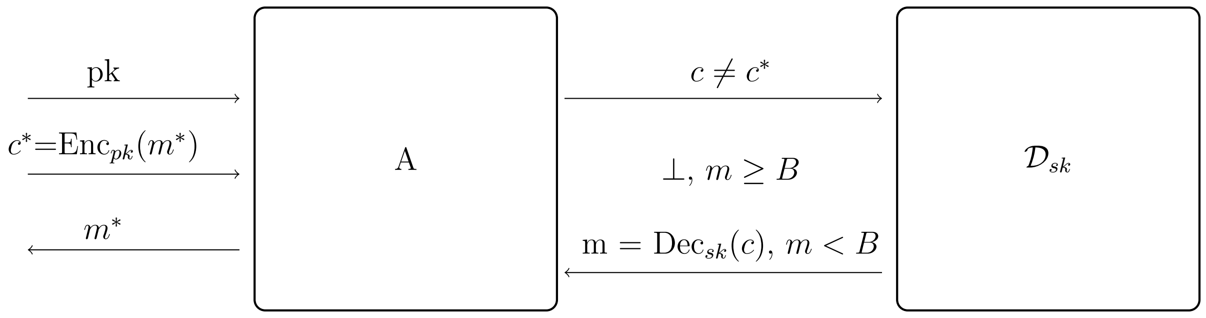

Figure 2: Manger's attack.

$$\begin{aligned} m &= mQ + q \cdot (qInv \cdot (mP - mQ) \pmod{p}) \\ m' &= mQ + q \cdot (qInv \cdot (mP' - mQ) \pmod{p}) \\ m - m' &= q \cdot (qInv \cdot (mP - mP') \pmod{p}) \\ &= q \cdot F \end{aligned}$$

In the above equation we notice that $m - m'$ can be written as a product of q and some factor F (the latter is irrelevant). What matters is that $m - m'$ is a multiple of q. Now recall that also N is a multiple of q (it is $p \cdot q$). Consequently the greatest common divisor of $m - m'$ and N must reveal q: $gcd(m - m', N) = q$.

Example: Let's consider a small example where $N = 17947$, and $e = 3$. Suppose we have a ciphertext $c = 8363$. We send the ciphertext to our decryption oracle, to get a reference decryption, which turns out to be $m = 513$. We send the same ciphertext to our decryption oracle, but this time we induce a fault in exponentiation mod p as described above, and receive as result $m' = 15447$. With this we factor N by computing $gcd(m' - m, N) = gcd(14934, 17947) = 131$.

4.1.3 Plaintext recovery according to Manger

The maybe best known attack on RSA-OAEP is due to Manger [8]. In contrast to the key recovery attacks in the previous section, the aim of Manger's attack is to recover the plaintext (see Figure 2). The attack that Manger describes essentially provides a little bit more information to the adversary (than the standard IND-CCA game) in the form of error messages from the decryption oracle.

Such error messages are part of the (original) specification of OAEP decryption `RSAES-OAEP-DECODE` (see RFC 2437, Section 7, [5]).

Alg. C: RSA Decryption according to RFC 2437

Steps:

```
1. If the length of the ciphertext C is not k octets,
***output*** "decryption error" and stop.

2. Convert the ciphertext C to an integer ciphertext
representative c: c = OS2IP (C).

3. Apply the RSADP decryption primitive (Section 5.1.2)
to the private key K and the ciphertext representative
c to produce an integer message representative m:

m = RSADP (K, c)

If RSADP outputs "ciphertext out of range," then ***output***
"decryption error" and stop.
```

Algorithm C shows the first three steps of `RSAES-OAEP-DECRYPT` : the first step is a sanity check on the input (given as octet string). The second step is to convert the input to an integer. The third step applies textbook RSA decryption to that integer and verifies that the result is is smaller than the modulus N, which is equivalent to checking that m starts with a trailing 00 byte (as specified by OAEP encoding). If this condition is not fulfilled another error message is given, and it is this error message that Manger's attack exploits.

Manger's attack utilises this error message to check if $m < B = 2^{8(k-1)}$ (k is the number of octets of N, thus if $m < B$ then m starts with 00). We make the assumption that $2B < N$, which is usually the case because the size of most RSA moduli is an exact multiples of 8 bits, making N at least 128 times bigger than B. If this condition is not fulfilled the attack still works but with a few tweaks.

The attacker is given a c^* and wishes to recover the corresponding $m*$, but the decryption oracle will not directly decrypt c^*. Hence adversary exploits the homomorphic property of RSA, and conforms to the "rules" of the CCA game by supplying specific inputs (other than c^*) $f^e \cdot c^* \pmod N$ (these correspond to $f \cdot m^* \pmod N$) to the decryption oracle. The decryption oracle indicates if the supplied $f \cdot m^*$ is in the range $[0, B)$ or (B, N) modulo N. Hence it gives a mathematical relationship for m^*: for a fixed f not all values of m^* can produce $f \cdot m^*$ that are in the range. In the attack the adversary aims to reduce this range with successive oracle queries until just one value is left: m^*.

This happens in three steps, which we run through now.

Step1: We first have to find a ciphertext that exceeds B. Therefore we send values $f_1^e \cdot c \mod n$ to the oracle until it returns that $f_1 \cdot m^* \geq B \mod n$. We know not only have a ciphertext that causes the error, but we also know that the "final" f_1 will satisfy $f_1 \cdot m^* \in [B, 2B[$, which implies that $\frac{f_1}{2} \cdot m^* \in \left[\frac{B}{2}, B\right)$.

Step 2: We choose an f_2 just under $n + B$ by setting $f_2 = \left(\left\lfloor \frac{n+B}{B} \right\rfloor\right) \cdot \frac{f_1}{2}$. Hence $f_2 = \left\lfloor \frac{n+B}{B} \right\rfloor \cdot \frac{f_{1,i}}{2}$ which means that $f_2 \cdot m^*$ is $\in \left[\frac{n}{2}, n+B\right[$ and test it with the oracle. If the oracle returns $> B$ then we increase f_2 by setting $f_2 = f_2 + \frac{f_1}{2}$ until we find a value for which the oracle returns $< B$. When this happens, we know that the modular wrap around must have happened and thus that the value was actually $> N$. Hence we have found an f_2 such that $f_2 \cdot m^*$ is $\in [N, N+B)$, and this gives us some bounds on the range for the unknown m^* (as $f_2 \cdot m^*$ is $\in [N, N+B)$ implies that m^* is $\in [m_{min} = N/f_2, m_{max} = (N+B)/f_2)$

Step 3: It follows from the fact that the ciphertext c^* was correctly produced that the underlying message is in $[0, B)$. Clearly it is possible to "shift" this message along the number line such that it would lie in some other interval $[i \cdot N, i \cdot N + B)$ (i.e. for any i we can find an f_3 such that $f \cdot m^*$ would lie in such an interval). In fact we can also find for any i an f_3 such that $f_3 \cdot m^*$ lies in $[i \cdot N, i \cdot N + 2B]$. Finding such i and f_3 would be very useful because the oracle that we have will tell us whether $f_3 \cdot m^*$ is larger or smaller than $i \cdot N + B$ in such a given interval. The choice of f_3 and i works as follows. First we calculate $f_{tmp} = \lfloor \frac{2B}{m_{max} - m_{min}} \rfloor$, and use this value to fix some $i = \frac{f_{tmp} \cdot m_{min}}{N}$. Now we can get an $f_3 = \lceil \frac{i \cdot N}{m_{min}} \rceil$, for which we know that it will be in $[i \cdot N, i \cdot N + 2B]$. We send this $f_3^e \cot c$ to the oracle which will tell us if the corresponding $f_3 \cdot m^*$ is smaller or bigger than B. This gives us new bounds for m^*: if $f_3 \cdot m^* \geq B$, then one can set $m_{min} = \left\lceil \frac{i_k n + B}{f_3} \right\rceil$. Else, if it returns $f_3 \cdot m < B$, one can set $m_{max} = \left\lfloor \frac{i_k n + B}{f_3} \right\rfloor$. We repeat Step 3 until the interval for m^* only contains one value. As we are effectively doing a binary search, we know that the process will terminate in $log(N)$ steps.

Example: Consider the parameters $N = 3551$, $e = 5$, and $c = 888$, all represented in decimal.

In Step 1 of Manger's attack we derive the smallest power of two, f_1, that satisfies $f_1 \cdot m \geq B$ using appropriate Oracle calls. In this example this step will result in $f_1 = 4$.

In Step 2 of Manger's attack we start with a value $f_2 = \lfloor ((n+B)/B) * f_1/2 \rfloor$, such that $f_2 \cdot m$ is just less than $N + B$. We then increase the value of f_2 by $f_1/2$ until $f_2 \cdot m$ wraps around. In our example, in the first round, $f_2 = 14 * 4/2 = 28$ and $f_2 * m = 3444$

(no modular wrap around). Hence we try again with the value of $f_2 = 28 + 2 = 30$, leading to $f_2 \cdot m = 3690$, which does wrap around.

In Step 3, because $N \leq f_2 \cdot m < N + B$, we can divide by f_2 to get some bounds on m (i.e. an interval of where m must be in). The purpose of the third step is to systematically reduce the length of this interval until it has size 1. In each round of Step 3 we will pick multiples of m, f_3, such that $f_3 \cdot m$ is in the interval $[in, in + 2B)$, for some index i. These values roughly double after each round, and because the interval has roughly constant size, this will shorten the range in which m must be in. The calls of the oracle will determine if $f_3 \cdot m$ is in the first half of the interval, or in the second one, and we change the values of m_max, m_min accordingly. In this example, three rounds are needed. We start off with $[m_min, m_max) = [119, 126)$, the we derive $[119, 124)$, then, $[121, 124)$ and last $[123, 124)$. Thus the last interval only contains 123. The values of f_3 are $59, 90, 147$ (they roughly double).

Correctness: the example was generated with $d = 1373$. Using this private exponent to decrypt you can verify that $m = 123$.

4.2 Attacks using Timing Information

An attack that utilises timing information, which was originally proposed by Kocher [6], is described in a very readable way by Dhem et al. [3]. The attack assumes the modular exponentiation of an RSA decryption, i.e. $c^d \pmod N$, is implemented as per Alg. A where by the modular multiplications are implemented with the Montgomery multiplication method.

The timing of a Montgomery multiplication is per se constant, independently of the multiplier, the multiplicand and the modulus. However if the intermediate result of the multiplication is greater than the modulus, then an additional subtraction (called a reduction) has to be performed. For the attack we assume that we can perform the decryption as often as we like with arbitrary inputs. Our goal is to deduce the private key d by analysing timing information that we obtain by measuring response times of our oracle, and by using a 'simulation' of the oracle.

The attacker proceeds as follows. They randomly sample some ciphertexts (the precise number depends on how much other factors impact on the timings that we measure from our oracle, and thus often needs to be experimentally determined). The attack recovers the secret exponent bit by bit. We know that the most significant bit is always one. Any other bit can obviously only take the values 0 or 1. Thus we look at each of these possibilities, make a "key bit guess" for each, and use our 'simulator' to predict for the set of input ciphertexts if (or not) for a given ciphertext an extra reduction during the modular squaring operation in the **next** round of the for-loop occurs. We separate the timing measurements (that we obtained from the oracle from the set of ciphertexts) into two sets according to our prediction.

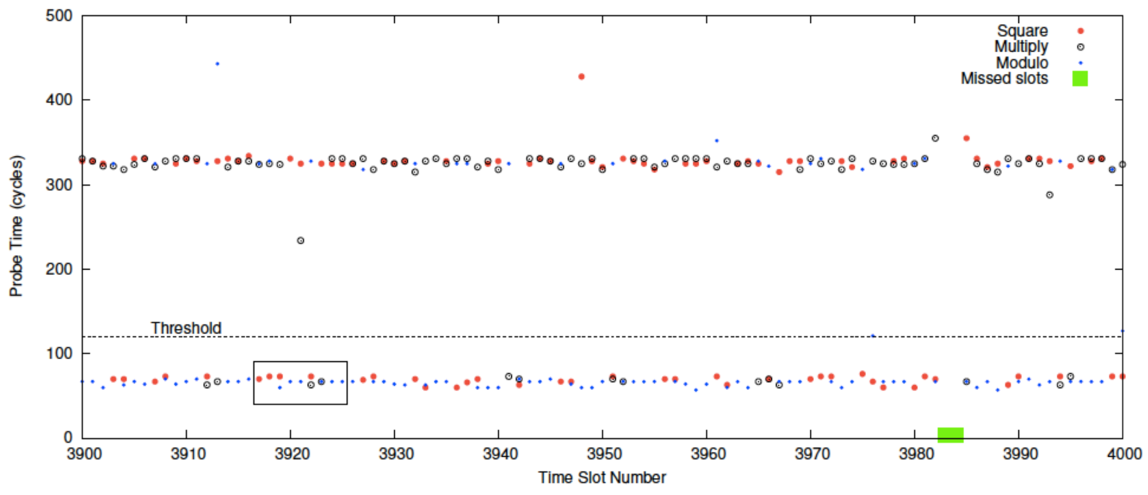

Figure 3: RSA cache timing trace from [10]

Now, let's assume that the true key bit was 0, and the key bit guess that we are looking at is also 0. In this case, our prediction matches what happens in the actual oracle. Hence there must be one set of measurements which should take on average a little bit longer to compute (and thus the corresponding response times should be a bit longer on average). We can "filter" out this timing difference by looking at the difference between average timings.

What happens though in the case where guess for the key bit is incorrect. So in our example this is that the actual key bit was 0 but we predicted it to be 1. In this case our simulator will no longer compute correspond to the actual device and thus when it comes to predicting if (or not) an extra reduction will occur in the squaring operation in the **next** round of the for-loop, our simulator can't make correct predictions. This means that we haven't got any meaningful separation of timings, and thus if we look at the average timing of the two sets that we create (based on the simulator predictions), this average is likely to be zero. An adversary who follows this process produces two average timings: the larger average difference corresponds to the correct key bit, and the smaller difference indicates the incorrect guess.

For historical reasons this kind of strategy is often referred to as a "Differential" attack, thus in the case timing information is used, it would be called a "Differential Timing Attack" in the literature. The crucial feature is that the attack works because it exploits tiny data-dependent differences by utilising many traces.

A "Simple Timing Attack" would, in contrast, utilise operation dependent differences,

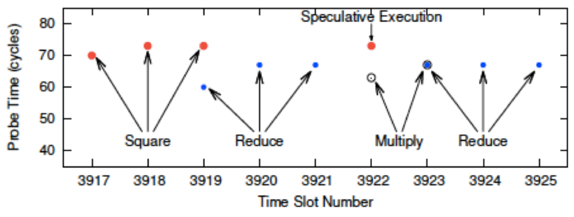

Figure 4: Zoom into RSA cache timing trace from [10]

and many of such attack work based on a single trace. For instance rather than utilising the overall execution time of the RSA decryption routine, we could try and insert some "software probes": a piece of code that times components such as the squaring, multiplication, and/or modular reduction functions. This idea can be utilised by measuring cache timings. In modern processors, there typically exists a hierarchy of different types of memory, and one element of these is a particularly fast type of memory, which is called the cache [1]. Usually processes share the cache memory. The idea behind so-called FLUSH+RELOAD cache attacks is to continually flush and reload specific cache lines. The cache lines are selected such that they hold variables of the victim process. For instance, if we know in which cache line the result of the square operation is held, we could flush the line, wait, and then reload the same line. If the victim process has executed the square, and thus touched this memory location, it would have caused this cache line to be reloaded. Thus the spy process's reload would be fast. If the victim process had not caused that cache line to be reloaded, the spy's reload would be slow. Yarom and Falkner [10] demonstrated this principle on an RSA implementation akin to the one we are considering, and Fig. 3 shows the cache timing trace of such an attack.

They provide a higher resolution picture of the boxed part of the figure, which we show in Fig. 4. What this picture shows is that their probing program shows that the victim program must have been requiring access to square, then reduce, then multiply, then reduce. They knew that the typical left-to-right algorithm for exponentiation was used, and in this algorithm, this sequence indicates that the corresponding key bit must have been 1.

4.3 Attacks using Power/EM Information

Power and EM information tends to be sampled at a high frequency and thus we can acquire many measurements values for a single execution of an algorithm (i.e. we get a "trace"). As a consequence it may be possible to observe features that can be directly

[1]There are in fact layers of cache memory, and we could spend a lot of time examing cache architectures. However for our purposes it suffices to known that typically there is something called last-level cache, which is shared among processes.

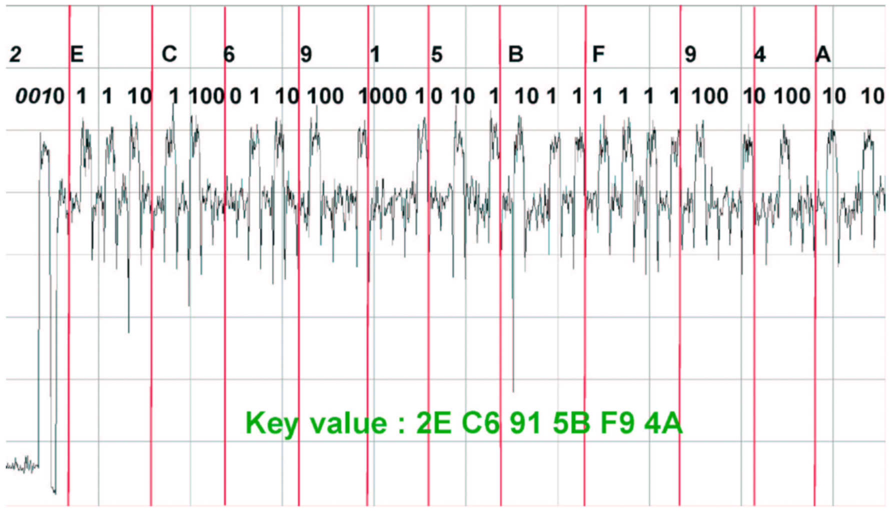

Figure 5: RSA trace

mapped to the flow of the execution of an algorithm. For instance, in the left-to-right exponentiation that we assume is implemented in a typical RSA decryption, there is a key dependent conditional branch (if and only if a key bit is 1 the multiplication is executed). This feature can be visible in power traces as first demonstrated by Kocher et al. [7] and shown in Fig. 5.

In the previous section we used timing information to distinguish key bits and thereby recover the decryption key. The exact same process also works with other types of side channel information such as power or electromagnetic traces. Like before, the adversary would select a set of ciphertexts (whereby the size of the set requires some consideration), and then build a simulator that predicts some property that is then tested for "consistency" with data obtained from the actual device/oracle under attack.

For instance the property could be the value of a specific bit of the result of squaring in the **next** iteration of the for loop. Based on this prediction the power traces sampled from the ciphertexts can be split into two sets (one where the value of the bit is predicted to be 1 and one where it is predicted to be 0). The difference in average power consumption is our distinguisher as in the timing attack example. We do this for both key bit guesses and thus end up with two difference traces. One of the key bit guesses must be correct and thus at some point in the power trace the adversary correctly predicts (part of) the behaviour of the device and this must result in a better separation between the sets (than in the case of the incorrectly predicted bit).

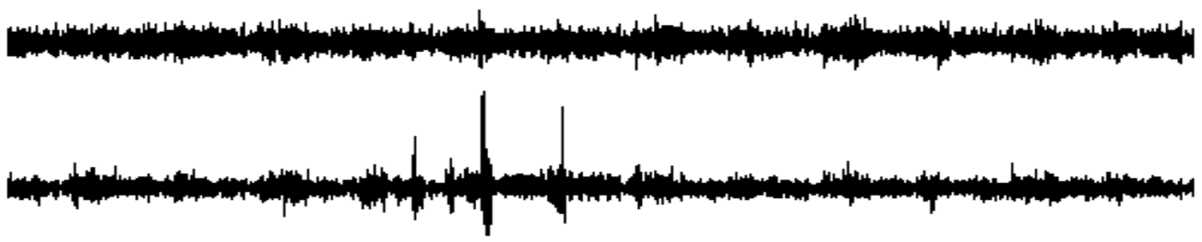
Figure 6: RSA Correlation traces (taken from [9])

One could argue that in case of a multiplication, basing predictions on a single bit may not be very effective (a typical hardware multiplier may operate on 32 bits). Hence another option, which is very frequently used in practice, is to predict the Hamming weight (i.e. number of 1 bits) of an intermediate value. This means a minor adjustment to the distinguisher is required: Hamming weight values are not just 0 and 1 thus it would make little sense to proceed as before (i.e. separate traces into two sets, and use the difference of averages as distinguisher). Instead, a much more intuitive strategy is to **correlate** the Hamming weight predictions with the power traces (each point is treated independently). This will result in two correlation traces: one for each key bit guess. The trace that somewhere shows some large correlation features will indicate the correct key guess, as demonstrated first by by Messerges et al. [9] and shown in Fig. 6 (the second trace indicates the correct key guess).

4.3.1 Other averaging attacks

Messerges et al. [9] describe another two simple attack ideas that may be applicable to RSA-OAEP. They both take advantage of the fact that we may be able to persuade the oracle to make some calculations with a known exponent. We discuss them in turn.

One idea they describe requires to be able to acquire multiple traces from a known exponent (of roughly the same length as the unknown one) and multiple traces from the unknown exponent. The premise of the attack is that if we calculate the average over many traces for both exponents, then the data-dependent portion of their power leakages will be similar, but there will be a difference because of the different exponent bits. The authors provide a working example of this technique, which we show in Fig. **??**

The other idea that Messerges et al. [9] describe relies on the ability to supply many known exponents to the oracle and obtain power traces from using these exponents on a single fixed data value. The known exponents are chosen in the same way as one proceeds in the "differential" attacks that we discussed before. The adversary starts from the most

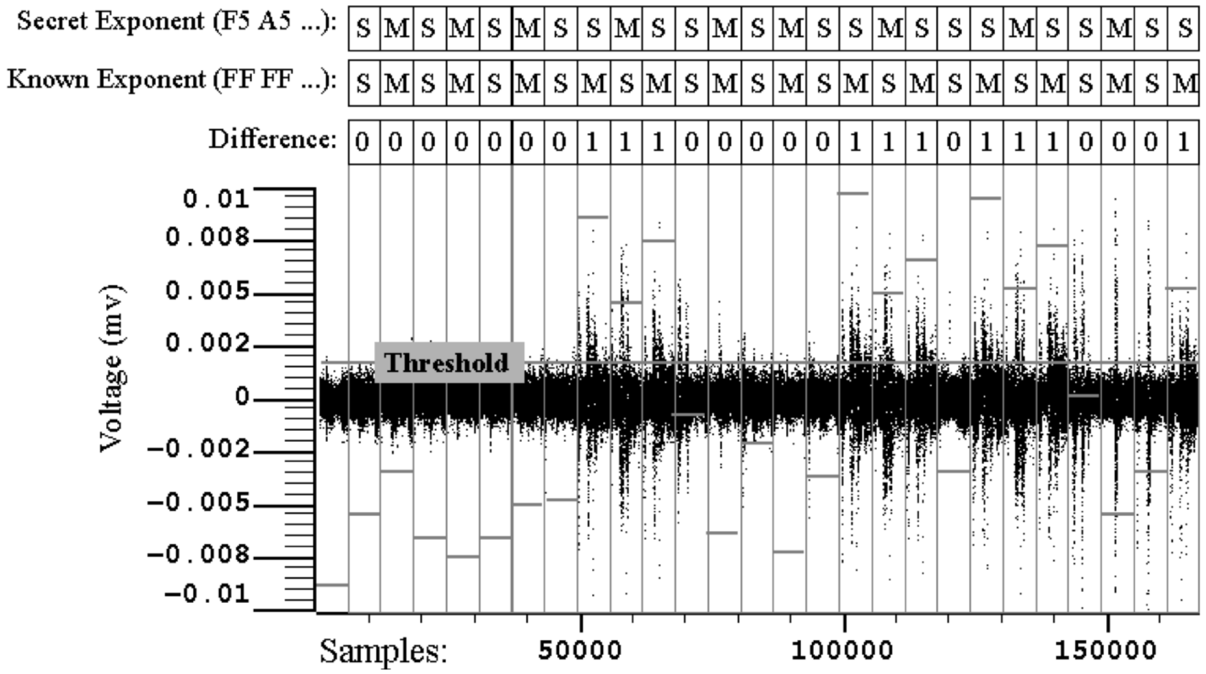

Figure 7: RSA traces from known/unknown exponent (taken from [9])

significant exponent bit and then recovers the next key bit. There are two options only, hence two known exponents are possible that are identical in the first known bits, then differ in a specific bit (the one that is being recovered) and the remaining yet unknown key bits can be random). A distinguisher is formed by subtracting the power trace from a key guess from the power trace corresponding to the unknown key. The key guess that shows a zero difference trace for "longer" indicates the correct key guess, see Fig. 8. In this attack, we are using the device to, in some sense, produce a prediction for us. In latter papers that idea is further developed and becomes known as "(online) templating".

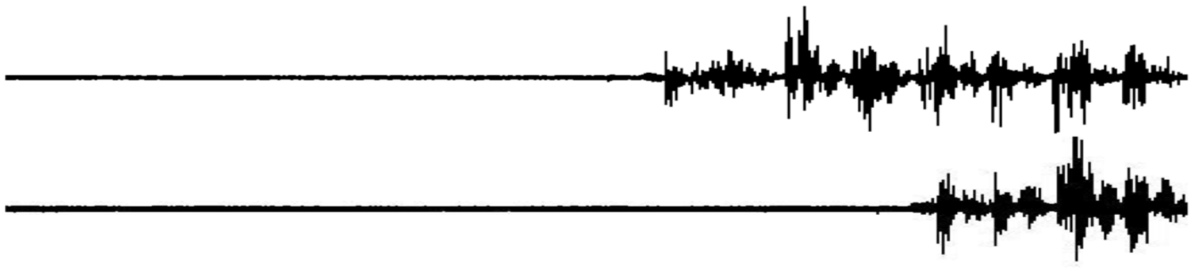

Figure 8: RSA traces from known vs fixed exponent (taken from [9])

5 Food for thought and Exercises

We looked at examples of timing attacks, but considered them all for RSA. Surely though the principle is much more widely applicable: the lab worksheet for this week suggests to implement such a strategy to recover PIN numbers.

Task A: Implement the PIN timing attack

Attend the lab session and implement the lab exercise that aims at recovering a PIN from timing information.

With regards to RSA, we illustrated the CRT based fault attack on a simple example, with numbers from a CRT calculation that I took from a website.

Task B: Fault attacks using CRT

Navigate to `https://www.di-mgt.com.au/crt_rsa.html` and use the numerical values to construct two further examples for this kind of fault attack. The first one should assume that an error was introduced in the exponentiation modulo p. The second one could assume a fault elsewhere, e.g. q^{-1} might be faulty.

We had a first look at various attack techniques that target the implementation of RSA (even when used in a secure form such as with OAEP). We considered a number of different attack vectors using different types of extra information or adversarial capabilities. The point of this is for you to realise that depending on circumstances, an adversary can choose from a rather wide range of known attack vectors.

Task C: Consider further attack vectors

What other sources of side channel information might be available in practice? Use up to an hour to research further sources of side channel information and consider if they might lead to attacks that could be applicable to RSA-OAEP.

Our investigations mainly touched on attacks: what mitigation strategies can you imagine could be implemented in practice?

Task D: Consider mitigation strategies

Take up to one hour to research/consider simple mitigation strategies.

We discussed attacks based on observing side channel information as well as attacks that require the adversary to actively induce faults. This suggests that implementation attacks can be categorised (like traditional cryptanalytic attacks) into active and passive attacks.

Task E: Categorise attacks

For the attacks that we investigated, consider whether they are active or passive. Also how invasive are different types of attacks?

Suppose you are asked in a professional capacity to comment on the applicability of a generic fault attack and a differential timing attack on an implementation of a system that is based on El Gamal rather than RSA.

Task F: El Gamal

Explain the potential for a generic fault attack and a differential timing attack on El Gamal. How might such attacks differ w.r.t the case of RSA?

Imagine you read a research paper that suggests a larger key (e.g., 2048-bit rather than 1024-bit) could help to prevent a timing attack on RSA.

Task G: Impact of key size

Explain whether and why you think a longer key is a mitigation against timing analysis.

6 Quiz

If you have worked through these notes and spent some time on the tasks, you should be able to answer the following questions.

1. Name at least three sources of side channel information. For each source, explain its' origin, how it can be measured, and how it may be utilised in an attack.
2. Explain, informally, the principle of a differential timing attack.
3. Give one example of a side channel attack that can reveal an RSA decryption key with a single side channel observation (trace). Describe in detail how this attack would work.
4. Give one example of an active implementation attack and one example of a passive implementation attack.
5. Explain how the original IND-CCA game captures (or not) sources of side channel information.

References

[1] Dan Boneh. Twenty years of attacks on the rsa cryptosystem. *NOTICES OF THE AMS*, 46:203–213, 1999.

[2] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In *Advances in Cryptology - EUROCRYPT '97*, pages 37–51, 1997.

[3] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestre, Jean-Jacques Quisquater, and Jean-Louis Willems. A Practical Implementation of the Timing Attack. In *CARDIS '98*, number 1820 in Lecture Notes in Computer Science, pages 167–182. Springer, 1998. Available online at `http://www.dice.ucl.ac.be/crypto/techreports.html`.

[4] Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In *Smart Card Research and Advanced Applications VI*, pages 159–176, Boston, MA, 2004. Springer US.

[5] B. Kaliski and Staddon J. Pkcs 1: Rsa cryptography specifications version 2.0, 1998.

[6] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO '96*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer, 1996.

[7] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[8] James Manger. A chosen ciphertext attack on rsa optimal asymmetric encryption padding (oaep) as standardized in pkcs #1 v2.0. In *Advances in Cryptology — CRYPTO 2001*, pages 230–238. Springer Berlin Heidelberg, 2001.

[9] Thomas S. Messerges, Ezzy Dabbish, and Robert Sloan. Power analysis attacks of modular exponentiation in smartcards. volume 1717, pages 144–157, 1999.

[10] Yuval Yarom and Katrina E. Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. In *Usenix Security*, 2014.