Side-channel attacks

Jordi L. Vermeulen
Studentnr. 3835634

Abstract

Side-channel attacks aim to retrieve secret data from a cryptographic system by observing factors outside the normal computation. If no care is taken, side-channel attacks can be used to compromise virtually any mathematically secure system. We give an overview of some types of side-channel attacks, as well as examples of how they target modern cryptographic systems in use today. We go on to survey possible countermeasures against such attacks. Note that this paper is in no way meant to be comprehensive, and that further research should be done before implementing a cryptographic system.

1 Introduction

Many cryptographic systems are assumed to be mathematically secure when the key is sufficiently large. Brute force attacks are assumed to be computationally infeasible, while other approaches don't do enough to abate this.

While it is good to have a system that is mathematically secure, this alone does not tell the whole story. There are attacks that aim not at the mathematical properties of the cryptographic system, but at implementation, hardware, electromagnetic radiation, timing and even sound. The generic name for such a method of attack is a *side-channel attack*. The name stems from the notion that the attack does not target the primary channel (the input and output of the cryptographic system, knowledge about the system itself, protocols used, etc.), but any other channel of information that might reveal secret data.

In this paper, we give an overview of three well-known and well-studied types of side-channel attacks: attacks that target power consumption, attacks that target timing and attacks that target faulty computations. For all three, we give examples of possible methods of attack, as well as suggestions for mitigating them.

2 Power analysis

Power analysis attacks are based on the notion that power consumption of cryptographic hardware is not constant during execution [15]. There are two types of power analysis: simple power analysis recovers information directly by observation of the power consumption of a device [8]; differential power analysis recovers information by exploiting statistical correlations between the power consumption of the hardware and the data being operated on [9]. In this section, we will explore both of these concepts in the context of RSA and AES. Following this, we will look at some ways to prevent the secret key from being recovered through these methods.

2.1 Simple Power Analysis

With simple power analysis, we can attempt to retrieve information directly from the power consumption of the device. The reason this is possible is that different operations consume different amounts of power, such that any conditional branches that depend on secret data potentially leak information about that data. A common example used for illustration is the RSA public-key cryptography system [12]. In RSA, decryption is performed by exponentiation of the ciphertext with the secret key. A fast and straightforward way to do this is by employing *exponentiation by squaring* (see Algorithm 1).

Algorithm 1 Pseudocode for exponentiation by squaring, with base C, exponent d and modulus n.

function EXPONENTIATION-BY-SQUARING(C, d, n)
  $result \leftarrow 1$
  while $d > 0$ **do**
    if $d \bmod 2 == 1$ **then**
      $result \leftarrow result \cdot C \pmod{n}$
    end if
    $C \leftarrow C \cdot C \pmod{n}$
    $d \leftarrow \lfloor d/2 \rfloor$
  end while
  return $result$
end function

The problem with this algorithm is that it has a conditional branch dependent on the secret key: when a bit in the key has the value of 1, it triggers an extra multiplication. Figure 1 shows a section of a power analysis of an

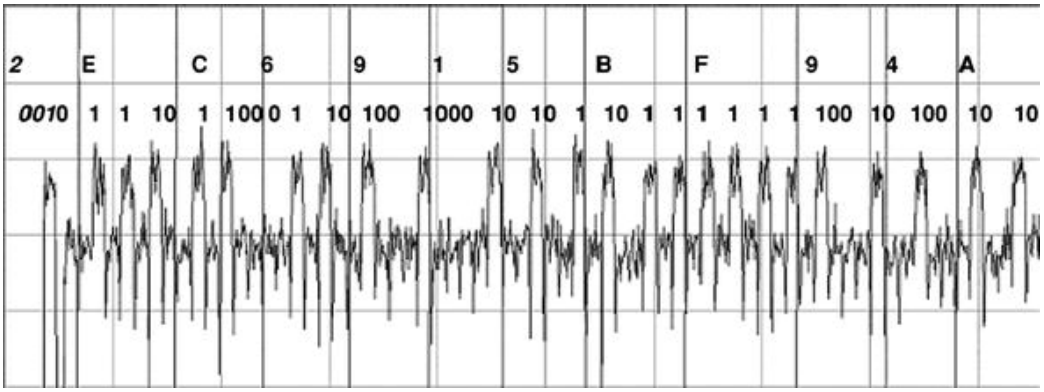

Figure 1: Simple power analysis of part of an RSA decryption operation. Original image by Joye and Olivier [5].

RSA decryption. We can see that there are obvious peaks in the power consumption. This is because the step with multiplication consumes slightly more power than the step without. This makes it possible to recover the key by simple power analysis *through only one decryption operation*.

2.2 Differential Power Analysis

Simple power analysis works well on cryptographic algorithms that have a strong correlation between the values of secret data and the power consumption of the hardware. Such a strong correlation is, however, not always present. In many such cases, differential power analysis can be of use. Instead of looking for a direct relation between secret data and power consumption, we look instead at variance in power consumption over many iterations of the algorithm. We then try to infer the secret data statistically.

Kocher et al. [9] give an example of how differential analysis can be used to retrieve an AES-128 key. The look at the first S-box lookup, $S[k_i \oplus n_i]$, where n_i and k_i are the i^{th} byte of the input and key, respectively. For many random plaintexts, they collect a power trace. Then, for each possible value of k_i $(1 \ldots 256)$, they examine the result of the S-box lookup, splitting the results on the least-significant bit. When one calculates the difference of the averages of the two resulting sets, the correct value for k_i will show large spikes. Because this method averages out many measurements, it is extremely insensitive to noise, making it all the more potent.

2.3 Countermeasures

One can employ a variety of measures to prevent secret data leaking out through power consumption. For the case of simple power analysis, it is generally enough to ensure no conditional branches depend on secret data. For instance, one can prevent simple power analysis on RSA decryption as mentioned above by adding dummy operations that ensure all code paths perform the same computations. Alternatively, one can employ a different method of exponentiation, called the *Montgommery Powering Ladder* [6] (see Algorithm 2).

Algorithm 2 Pseudocode for the Montgommery Powering Ladder, with base C, exponent d and modulus n. d_j is the j^{th} bit of d.

```
function MONTGOMMERY-POWERING-LADDER(C, d, n)
    R_0 ← 1; R_1 ← C
    for j = t − 1 downto 0 do
        if d_j == 0 then
            R_1 ← R_0 · R_1 (mod n)
            R_0 ← R_0 · R_0 (mod n)
        else
            R_0 ← R_0 · R_1 (mod n)
            R_1 ← R_1 · R_1 (mod n)
        end if
    end for
    return R_0
end function
```

Preventing differential power analysis is more complicated. For public-key algorithms, it is possible to *blind* the secret data. In the case of RSA, we choose a pair (v_i, v_j), with $v_i = (v_j^{-1})^e \bmod n$, where e is the public key. Before decrypting C, we then multiply it by $v_i \pmod n$, and before returning the result of the decryption, we multiply by $v_j \pmod n$. (Verifying that this does indeed result in the original plaintext is left as an exercise to the reader.) Because (v_i, v_j) is chosen randomly for every encryption, it is hard to combine the statistical data gathered over multiple iterations, making it difficult to extract the secret key [9]. For symmetric algorithms like AES, we can employ a similar technique, called *masking*, in which parts of the secret data are randomised, only to be returned before calculation of the final result.

3 Timing

Timing attacks target any correlation between secret data and the time it takes to perform a computation in order to retrieve said secret data. The computation being timed can vary from a single operation [14], to the entire cryptographic function [7]. It is even possible to perform a timing attack against a server on a network [4]. In this section, we will have a closer look at conventional and cache-timing attacks, how they recover secret data and how they can be prevented.

3.1 Conventional timing

Conventional timing attacks directly measure the time taken to perform a cryptographic computation. Kocher [7] shows how these measurements can be used to obtain the secret key in public-key systems such as RSA, Diffie-Hellman and DSS. Recall Algorithm 1, exponentiation by squaring. Clearly, for each iteration, the amount of computation performed differs depending on the value of the current bit of the exponent and the value of the ciphertext. The attack works by observing the exponentiation of multiple known ciphertexts C, called samples. Each timing observation can be seen as a function $T = e + \sum_{i=0}^{w-1} t_i$, where t_i is the time required by the step for bit i and e collectively represents all the time spent on calculations other than the multiplication and squaring. Given a guess of the first b bits of the exponent, it is possible to find $\sum_{i=0}^{b-1} t_i$ for each sample. If the guess is correct, subtracting from T yields the time taken by the remainder of the computation: $e + \sum_{i=b}^{w-1} t_i$. The variance of this expression over all samples is expected to decrease for each correctly guessed bit, while it increases for each incorrectly guessed bit. Kocher shows that the number of samples required to successfully retrieve the secret key is proportional to the length of the key.

3.2 Cache-timing

A cache-timing attack aims to recover secret data by monitoring cache performance. The basic notion is that a cache hit will dramatically speed up most operations (or, equivalently, a cache miss will slow them down). If the pattern of hits and misses is dependent on the secret data, it is possible to retrieve that data by looking at the performance characteristics of the operation. Bernstein [1] shows how this method can be applied to retrieve

AES keys when the plaintext is known. He observes that at the start of the algorithm, there is a table lookup $S[k_i \oplus n_i]$, which is the S-box substitution of each byte in the input, where n_i and k_i are the i^{th} byte of the input and key, respectively. If we assume that the total running time of the AES encryption is well-correlated with the time needed to perform this table lookup and is independent of the key used, we can statistically determine the key by comparing observed timings with timings calculated in advance for some known key. Bernstein shows that these assumptions are correct, and is able to derive a 128-bit AES key from timing information.

Another cache-timing attack by Yarom and Falkner [14] shows that it is possible to retrieve up to 98% of the bits in an RSA key from a single encryption through a method called FLUSH+RELOAD. It is based on a spy program operating on the same machine, which selectively causes lines of memory to be removed from the cache. It can then monitor the time at which the line is returned to the cache, which leaks information about data access by the encryption program. It operates on L3 cache, meaning it does not need to share a core with the encryption program and that it can be used across VMs.

3.3 Countermeasures

Many solutions to timing attacks have been proposed. For conventional timing attacks, it is recommended to employ blinding (as discussed in Section 2). This way, any data leaked reveals nothing about the secret key. Alternatively, it is possible to require all cryptographic operations take as much time as the longest one, but this has an obvious adverse effect on performance [4].

To prevent cache-timing attacks, Page [11] and Bernstein [1] advocate against the caching of S-boxes, or mitigating them altogether in future cryptographic standards. For AES, Osvik, Shamir and Tromer [10] advocate data-oblivious memory access patterns, disabling cache sharing or implementing a static cache. Wang and Lee [13] suggest to redesign the current method of caching, proposing a *random permutation cache*, which randomises the cache allocation so as to not reveal which parts of the cache correspond to what data. It is worth noting that hardware implementations of cryptographic algorithms generally lack any form of caching, meaning that only software implementations meant for general-purpose computers are vulnerable to this type of attack.

4 Fault analysis

Fault analysis attacks attempt to retrieve secret data from the result of faulty computations. These computations may come about through faulty hardware or deliberate tampering with the device or software. The concept of fault analysis attacks was introduced by Boneh, DeMillo and Lipton [3]. In this section we look at how fault analysis can recover the secret key used in RSA and Rabin signatures [3] and how differential fault analysis can be used to find the secret key of DES [2].

4.1 Conventional fault analysis

Conventional fault analysis aims to retrieve secret data by analysing the result of faulty encryptions. We will illustrate how this may be done by looking at RSA signatures [12].

RSA signatures are made by RSA encryption of the hash of a message with a secret key d. Integrity of the signature can be checked by decrypting it with the public key e and checking that the resulting hash matches the message. Recall that RSA encryption is defined by the function $E(x) = x^d \bmod n$, where n is the product of two primes p and q. Only the holder of the secret key knows the factors of n. If an attacker is able to factor the modulus n, it is equivalent to breaking the algorithm, because it allows him to calculate the secret key d from the public key e.

Knowing the factors of n allows the party signing a message to speed up the computation of $x^d \bmod n$ by use of the Chinese Remainder Theorem (CRT). It allows the computation to be split into $E_1(x) = x^d \bmod p$ and $E_2(x) = x^d \bmod q$. This can then be used to calculate $E(x)$ when we know two numbers a and b such that

$$\begin{cases} a \equiv 1 \pmod{p} \\ a \equiv 0 \pmod{q} \end{cases} \quad \text{and} \quad \begin{cases} b \equiv 0 \pmod{p} \\ b \equiv 1 \pmod{q} \end{cases}$$

When we know these numbers, we can find the final result by calculating $E(x) = aE_1(x) + bE_2(x) \pmod{n}$.

Now suppose that we have a pair of signatures for the same message: E, the correct signature; and $\hat{E}$, a signature for which a fault was introduced into the computation of $\hat{E}_1$. We find

$$E - \hat{E} = (aE_1 + bE_2) - (a\hat{E}_1 + b\hat{E}_2) = a(E_1 - \hat{E}_1)$$

Boneh, DeMillo and Lipton note that it is unlikely that $E_1 - \hat{E}_1$ is divisible by p, in which case

$$gcd(E - \hat{E}, n) = gcd(a(E_1 - \hat{E}_1), n) = q$$

We have now found a factor of n, making it trivial to find the other. Boneh, DeMillo and Lipton go on to show that similar techniques can be used to break Rabin signatures, Fiat-Shamir and Schnorr identification, as well as implementations of RSA that do not employ CRT.

4.2 Differential fault analysis

Biham and Shamir [2] expand on the notion of fault analysis, making it applicable to symmetric cryptographic systems as well. They call their approach *differential fault analysis*. Their attack works under the assumption that it is possible to induce a fault in the computation at some random point. They explain how their attack would work on DES.

The attacker encrypts the same plaintext twice, once with and once without an induced error. He then tries to identify the round in which the fault occurred by looking at the difference between the two obtained ciphertexts. By analysing the way bits influence the final result in the last three rounds, he can discard many possible key values. Empirical analysis shows that it is possible to retrieve the last round key given about 50 to 200 ciphertexts. The last round key contains 48 of the 56 bits in the full key, making it trivial to test all remaining possible values to find the full key.

To use the same attack against an algorithm like 3DES, however, it is more practical to take a different approach. Instead of testing the remaining keys, we "peel off" the last round of the algorithm, and then proceed to attack the previous round, until all rounds have been broken. In this way it is possible to simply iterate the attack until sufficient rounds have been removed.

4.3 Countermeasures

One way of defending against fault analysis attacks is to check the result of the cryptographic function before outputting it. In the case of RSA, the device should simply check that decrypting with the public key yields the desired result. For some other cryptographic systems, it is not enough to check the result (e.g. multi-round authentication schemes, where faults may be induced between rounds). In this case error detection bits may be employed. For example, space may be reserved for storing CRC values of data in certain registers, so that the integrity of its content may be verified.

5 Conclusions

We have seen three types of side-channel attacks: power analysis, fault analysis and timing attacks. All these attacks have the potential to break many modern cryptographic systems. As such, great care must be taken when implementing any cryptographic system; although it may be mathematically secure, implementations and hardware may expose weaknesses that make it trivial to recover secret data. Countermeasures include, but are not limited to:

- Ensure constant-time and constant-power execution of the algorithm.
- Employ blinding and masking where appropriate.
- Employ a static cache, or disallow caching altogether.
- Ensure that memory access is data-oblivious.
- Redesign the traditional caching mechanism in such a way that cache performance gives no information about secret data.
- Verify results before outputting them.
- Implement integrity checks on vital pieces of memory.

This list is by no means exhaustive, but implementing a selection of these features will greatly assist in securing a cryptographic system against side-channel attacks.

References

[1] D. J. Bernstein. Cache-timing attacks on AES. Technical report, University of Illinois at Chicago, 2004.

[2] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology – CRYPTO '97*, volume 1294 of *LNCS*, pages 513–525. Springer, 1997.

[3] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology – EUROCRYPT '97*, volume 1233 of *LNCS*, pages 37–51. Springer, 1997.

[4] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[5] M. Joye and F. Olivier. Side-Channel Analysis. In *Encyclopedia of Cryptography and Security*, pages 1198–1204. Springer, 2011.

[6] M. Joye and Y. Sung-Ming. The Montgommery Powering Ladder. In *Cryptographic Hardware and Embedded Systems – CHES '03*, volume 2523 of *LNCS*, pages 291–302, 2003.

[7] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113, 1996.

[8] P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397, 1999.

[9] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to Differential Power Analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[10] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006.

[11] D. Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, 2003.

[12] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[13] Z. Wang and R. B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, 2007.

[14] Y. Yarom and K. Falkner. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. Cryptology ePrint Archive, Report 2013/448, 2013.

[15] Y. B. Zhou and D. G. Feng. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. Cryptology ePrint Archive, Report 2005/388, 2005.