1

Timing Attacks

Daniel PAGE
University of Bristol, United Kingdom

Set within the context of cryptographic engineering, timing attacks are a class of side-channel attack: if the execution latency of some target operation depends on security-critical data, this may allow an attacker to leverage passive execution latency measurements in any attempt to recover said data. Timing attacks have a rich history, and, in part because variation in execution latency can arise from a wide range of sources, remain surprisingly stubborn. Although a known problem, they remain far from a solved problem as evidenced by related instances continuing to appear more than 20 years since initial publications on the topic. The aim of this chapter is to offer an introduction to that topic, covering *both* a set of foundational concepts from a more theoretical perspective *and* a set of example attacks and their mitigation (via countermeasures) from a more applied perspective. As a result, the content should enable you to understand and apply the concepts covered in a broader context, for example, within your own work, and beyond the specific, limited set of examples used.

1.1. Foundations

In this section, we focus on explanation of the central concepts. The aim, in fairly non-technical terms, is to first explain *what* a timing attack is, and then establish a terminology and model for characterizing and reasoning about them. Doing so offers a foundation, which we will then build on in section 1.2 through an investigation of more concrete, example attacks.

Embedded Cryptography 1,
coordinated by Emmanuel PROUFF, Guénaël RENAULT,
Matthieu RIVAIN and Colin O'FLYNN.

```
1  algorithm BUBBLE(X, n) begin
2      do
3          f ← false
4          for i = 1 upto n − 1 do
5              if X_{i−1} > X_i then
6                  X_{i−1} ↔ X_i
7                  f ← true
8              end
9          end
10     while f = true
11 end
```

(a) A more abstract description of bubble sort, using pseudo-code

```
1  void bubble( int* X, int n ) {
2    bool f;
3
4    do {
5      f = false;
6
7      for( int i = 0; i < n; i++ ) {
8        if( X[ i - 1 ] > X[ i ] ) {
9          int t           = X[ i - 1 ];
10             X[ i - 1 ] = X[ i     ];
11             X[ i     ] = t;
12
13         f = true;
14       }
15     }
16   }
17   while( f == true );
18 }
```

(b) A more concrete description of bubble sort, using C source code

Figure 1.1. *Two descriptions of bubble sort*

1.1.1. *Execution latency in theory*

DEFINITION 1.1.– Given a computational process P, the term *execution latency* is a measure of the delay between P starting and finishing execution (whether it completes normally or is terminated abnormally).

We intentionally leave the *unit* of measurement undefined. That is, within different contexts it might be appropriate to use different units, for example, an abstract unit such as computational steps, a concrete, human-oriented unit such as seconds, or a concrete, machine-oriented unit such as clock cycles. If the unit relates to time in a meaningful way, then it is common to see execution *time* used synonymously with execution *latency*: both terms capture the idea that we measure how much time elapses between starting and finishing execution.

The bubble sort algorithm is conceptually simple, so is an appealing, generic example to use as a vehicle to explore this concept further. The algorithm, described in Figure 1.1(a) using pseudo-code, repeatedly iterates over an n-element input array X, comparing adjacent elements and swapping them in-place if they occur in the wrong order; this is repeated until the array is sorted, which we know is true when no more swaps are required.

So, given an input array, what is the execution latency of this algorithm? That is, how much time elapses between starting (using X and n as input) and finishing execution (meaning a sorted X is produced as output) of the algorithm? Analysis of computational complexity (or, more specifically, time complexity) offers one

approach to providing an answer. Doing so gives a result in terms of abstract computational steps, focused on the problem size n. By using such analysis, we conclude that in the worst case *and* average case, the algorithm will perform $O(n^2)$ comparison operations and $O(n^2)$ swap operations; in the best case, the algorithm will perform $O(n)$ comparison operations and $O(1)$ swap operations.

DEFINITION 1.2.– The execution latency of a computational process P is termed *data dependent* if it depends on, and so may vary as a result of, the input data. Otherwise, it is termed *data independent*: this implies that the execution latency is the same, irrespective of the input data for every execution of P.

Two facts should be clear from our analysis of bubble sort. First, and more obviously, the *length* of X impacts the execution latency: in all cases, the number of steps will be greater for an X with many elements (i.e. larger n) than it will for an X with few elements (i.e. smaller n). Second, and less obviously, however, the *content* of X *also* impacts the execution latency: as suggested by the differing best and worse cases, the number of steps will be less for an X which is already (close to being) sorted than it will for an X, which is not. Either way, execution latency of the bubble sort algorithm would therefore be classified as data dependent.

1.1.2. *Execution latency in practice*

DEFINITION 1.3.– In concrete terms, data-dependent execution latency arises from (at least) the following cases: (1) *which operations are executed*, as determined by control-flow decisions made during execution; and (2) *how operations are executed*, or, put another way, what the execution latency of each executed operation is and whether or not it is data dependent.

Although instances of the first case can be identified within an abstract algorithm, instances of the second case are dependent on concrete details of an associated implementation and platform it is executed on. Conceptually at least, we could view this shift as including rather than (intentionally) excluding constant factors in analysis using computational complexity. Doing so is clearly imperative, because we ultimately aim to reason about concrete, real-world versus abstract, on-paper scenarios.

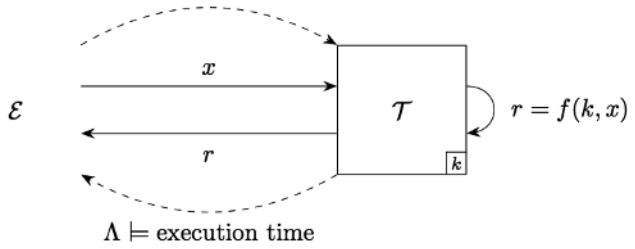
We can identify instances of both cases using Figure 1.1(b), which describes an implementation of bubble sort using C source code. For example, the `if` statement on line 8 represents an example of the first case: the flow of control is managed so lines 9–13 are either executed or not, depending on whether or not `X[ i - 1 ] > X[ i ]`. Likewise, the memory access implied by the expressions `X[ i - 1 ]` and `X[ i ]` on line 8 represents an example of the second case: the latencies of said accesses depend, for example, on the memory hierarchy (e.g. the

existence and state of any caches that exist within it). More generally, the platform will optimize instruction execution; it may do so for both memory access *and* computational instructions (for example, although this implementation does not include any, shift, multiplication, and division instructions sometimes exhibit data-dependant execution latency). An implication of this fact is that robust analysis of data (in)dependence cannot arise from the (static) source code alone. The Instruction Set Architecture (ISA) said source code ultimate targets will typically only capture functional properties: how instructions will be (dynamically) executed, and therefore the behavioral properties of the platform (i.e. a specific micro-architectural implementation of that ISA), *also* need to be assessed.

1.1.3. *Attacks that exploit data-dependent execution latency*

In a sense, bubble sort is as efficient as it can be: it only performs the steps required to produce a correct output from an associated input by exiting the outer-loop as soon as no more swaps are required. Adopting a perspective where efficiency is the metric of interest, such an approach is clearly positive. In fact, do *any* negatives stem from it? If we adopt a different perspective where security is the (or at least a) metric of interest, the answer is yes.

Put simply, the fact that execution latency is data dependent implies some form of correlation: measuring the execution latency will provide information, of some form, about the data it depends on. Set in some scenario where an adversary has access to execution latency measurements, this can be hugely problematic. Specifically, consider a scenario where the data involved are security critical, for example, but without loss of generality, some cryptographic key material. Using bubble sort as an example starts to become tenuous at this point, so instead consider the following generic attack model:



which helps fix various concepts and notation. $\mathcal{E}$ is using $\mathcal{T}$ to perform some computation f. Having first accepted an input x from $\mathcal{E}$, $\mathcal{T}$ computes and then provides $r = f(k, x)$ as output. $\mathcal{E}$ can also measure the execution latency associated with computation by $\mathcal{T}$, which we denote Λ. On the one hand, k *is* not and *should* not be known by $\mathcal{E}$. On the other hand, however, Λ may provide information about k if the execution latency of f is data dependent, and therefore may potentially be

exploited in an attack aiming to recover it. This is the crux of a side-channel attack based on execution latency, or, less formally, a *timing attack*. We say that information about the security-critical data is leaked (or is leakage) through a side-channel represented by execution latency. Such attacks are potent, because, as a result of exploiting the additional information afforded by the side-channel, they can often be significantly more efficient than alternatives (e.g. traditional cryptanalysis).

Now focused on the topic of side-channel attacks, we need to address two important points regarding terminology. First, in relation to data (in)dependence, the term data are qualified by the fact that we almost always mean security-critical data. There is limited value in learning information about data we already know, for example, and so no implication for execution latency depending on it. So, from here on, read "data" as "security-critical data" unless otherwise stated. Second, it is very common to see the term constant-time (respectively, variable-time) used as a synonym for data-independent (respectively, data-dependent) execution latency in this context (already ignoring the clash vs. $O(1)$ in computational complexity). We avoid these terms, deeming data (in)dependence more reflective of what is actually intended. For example, the execution latency of a given algorithm *may* vary and need not reflect any *specific* constant: we only demand that it does *not* vary, and so is (some) constant with respect to security-critical data involved. Likewise, use of the term constant-time suggests execution time (or latency) is the *only* pertinent issue; in reality, and more generally, we need to address the challenge of data-dependent resource use (*both* in terms of which resources are used, *and* the length of time they are used for). We can find examples of this fact investigated in Chapter 2.

Having presented the high-level concept, we now refine various lower level details of our attack model. We do so in a piecemeal manner below, framing the discussion around a set of concise definitions which, in combination, help to explain and characterize specific attacks (such as those used as examples in section 1.2).

DEFINITION 1.4.– The attack strategy employed by $\mathcal{E}$ may be described as *differential* or *non-differential*.

At a high level, a non-differential attack will typically use few (even just one) execution latency measurements in isolation. In contrast, a differential attack will typically use many execution latency measurements in combination, for example, through analysis of when and how they differ as the result of data-dependent behavior by $\mathcal{T}$.

DEFINITION 1.5.– The attack strategy employed by $\mathcal{E}$ may be assessed, where relevant, in relation to either *efficacy* and/or *efficiency*.

Although exceptions might exist, efficacy is normally a binary assessment: either the attack by $\mathcal{E}$ is successful with respect to some stated aim or not. In contrast, there

are many ways to assess efficient resource use by $\mathcal{E}$ during said attack. Focusing on time, for example, we might view the number of interactions with $\mathcal{T}$, the high-level, algorithm-related attack latency (e.g. stated in terms of computational complexity) or the low-level, implementation-related attack latency (e.g. stated in terms of wall-clock time), as indicative of how efficient $\mathcal{E}$ is.

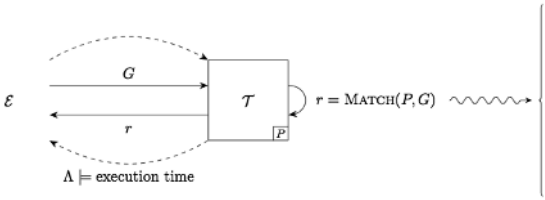
DEFINITION 1.6.– If the interaction between $\mathcal{E}$ and $\mathcal{T}$ is direct, the attack is described as *local*; otherwise, if the interaction is indirect, the attack is described as *remote*.

There are other, reasonable ways to capture a similar concept, for example, using physical proximity (implying physical access) as the measure of localness. We opt for the above instead, because $\mathcal{E}$ and $\mathcal{T}$ may be in close physical proximity but *still* physically or logically isolated from each other; this would imply an indirect form of interaction. Either way, two points are important. First, although the attack model above at least *suggests* that $\mathcal{E}$ and $\mathcal{T}$ interact remotely, this is *not* a limitation. For example, $\mathcal{E}$ may leverage physical access to some device $\mathcal{T}$ in order to mount a local attack; $\mathcal{E}$ and $\mathcal{T}$ may be software processes (co-)resident and so executing locally on the same platform; $\mathcal{E}$ and $\mathcal{T}$ may be software processes resident and so executing on different platforms, interacting remotely via an intermediate network. Second, if/when a remote attack is feasible, this is often viewed as an advantage. For example, such an attack typically removes the need for physical access to $\mathcal{T}$, avoids any reliance on additional equipment (for example, see power- or EM-based side-channels) and allows attacks to be applied at greater scale (in the sense there are likely to be more accessible instances of $\mathcal{T}$ to attack).

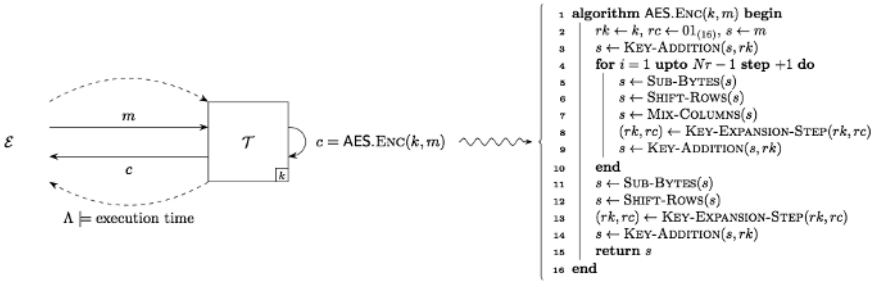
DEFINITION 1.7.– From the perspective of $\mathcal{E}$, computation by $\mathcal{T}$ may be described as *synchronous* or *asynchronous*.

The synchronous case suggests that $\mathcal{E}$ knows or even *controls* when $\mathcal{T}$ starts or finishes execution; in contrast, the asynchronous case suggests that $\mathcal{T}$ operates independently from $\mathcal{E}$ (and, as a result, is often described as free-running). In the asynchronous case, one typically attempts to identify some form of trigger or reference event that indicates the start and/or finish of execution. Such an event might be observable via a secondary side-channel, or simply observation of when and how $\mathcal{T}$ interacts with external parties (such as $\mathcal{E}$) or resources (such as memory or peripheral devices).

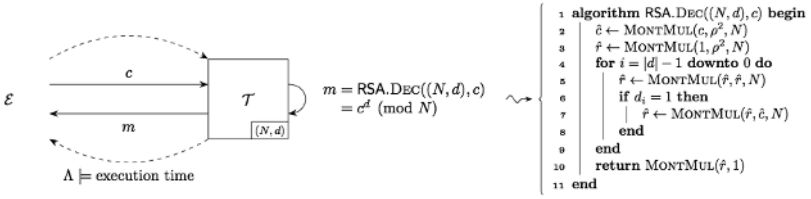
DEFINITION 1.8.– The execution latency measurements taken by $\mathcal{E}$ are characterized by their *accuracy*, that is, how close the measured and actual execution latencies are, and *precision*, that is, how close two execution latency measurements for the same computation are. The former case typically relates to systematic error, for example, due to the impact of *quantization*; whereas the latter case typically relates to random error, for example, due to the impact of *noise*.



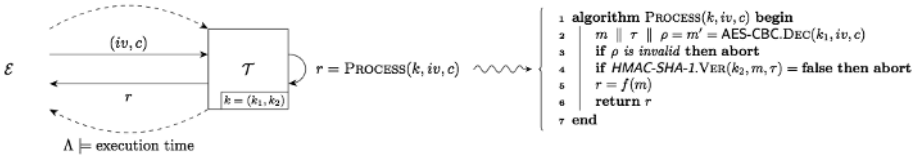
(a) The attack model for Example 1.1, in section 1.2.1.



(b) The attack model for Example 1.2, in section 1.2.2.



(c) The attack model for Example 1.3, in section 1.2.3.



(d) The attack model for Example 1.4, in section 1.2.4.

Figure 1.2. *Attack models relating to the examples presented in section 1.2*

We deem any formal exploration of what noise *is* beyond our scope; it suffices to understand that noise will degrade accuracy and precision, in the sense that execution latency of the same computation within different interactions between $\mathcal{E}$ and $\mathcal{T}$ may differ (or fluctuate). The term Signal-to-Noise Ratio (SNR) is often used to capture the relative strength of the signal of interest, that is, the actual execution latency, versus any noise that causes the measured execution latency to differ. Both statistical and non-statistical techniques can typically be harnessed in an attempt to reduce or even

remove noise, that is, yield higher SNR; examples include averaging and filtering. Doing so can be important, because, ultimately, SNR will typically have some impact on the efficiency and/or efficacy of most attacks.

1.2. Example attacks

In this section, we focus on application of the central concepts: the aim is to illustrate *how* timing attacks are mounted. Harnessing the foundation offered by section 1.1, we do so by considering a small set of examples each framed within a simple but plausible scenario; we structure each example in the same way by (1) defining the attack model, (2) analyzing the attack model, for example, to characterize the source of data dependence and then, finally, (3) presenting an attack based on said analysis.

1.2.1. *Example 1.1: an explanatory attack on password validation*

1.2.1.1. *Scenario*

Consider the attack model depicted in Figure 1.2(a): an adversary $\mathcal{E}$ interacts with a target device $\mathcal{T}$ that stores some embedded, security-critical data P (a password). When provided with input G (a guess at the password), $\mathcal{T}$ first performs the computation detailed by the algorithm MATCH (that is, string matching, or, more specifically password validation) and then responds with r as output. $\mathcal{E}$ can *also* measure the execution latency of computation performed by $\mathcal{T}$, so attempts to exploit this information with the aim of recovering P (implying, for example, it then gains access to $\mathcal{T}$).

1.2.1.2. *Analysis*

In the same way we reasoned about bubble sort, the MATCH algorithm is as efficient as it can be: as soon as it detects a mismatch between P and G, it immediately returns *false* rather than needlessly continuing. Such a mismatch can occur for two reasons. First, and resulting in execution of line 3, the mismatch might stem from the *length* of, that is, number of characters in, P and G: if $|P| \neq |G|$, then the two cannot match. Second, and resulting in execution of line 7, the mismatch might arise from the *content* of P and G: if $P_i \neq G_i$ for some i, then the two cannot match. This is termed early-abort behavior, and is possible when a special-case condition means the general-case algorithm need not be fully executed in order to provide correct output from the associated input. Viewed from the perspective of efficiency, this behavior is positive; when viewed from the perspective of security, the same behavior becomes negative; however, it means that the execution latency of MATCH is data dependent, that is, depends on P and G.

1.2.1.3. *Exploitation*

Assume, without loss of generality, that the set of valid characters is $A = \{\texttt{'a'}, \texttt{'b'}, \ldots, \texttt{'z'}\}$, and, specifically, that $P = \texttt{"pencil"}$, which means $|P| = 6$. The attack proceeds in two phases:

1) In this phase, the attack aims to recover $|P|$. To do so, we use a sequence of guesses:

$$G \in \langle \texttt{"a"}, \texttt{"aa"}, \texttt{"aaa"}, \texttt{"aaaa"}, \ldots \rangle$$

during interaction with $\mathcal{T}$, that is, the ith guess is $i + 1$ characters in length. The execution latency for guess $G = \texttt{"aaaaaa"}$ will be longer than for the others because $|P| = |G|$; this means execution for it will progress into the loop, rather than early-abort at line 3. As such, identifying this G recovers $|P|$.

2) In this phase, the attack aims to recover P iteratively, that is, character-by-character. To recover P_i in iteration $i = 0$, we use a sequence of guesses:

$$G \in \langle \texttt{"aaaaaaa"}, \texttt{"baaaaaa"}, \texttt{"caaaaaa"}, \ldots, \texttt{"zaaaaaa"} \rangle$$

during interaction with $\mathcal{T}$, that is, the ith character cycles through all possibilities, while the others remain fixed. The execution latency for guess $G = \texttt{"paaaaa"}$ will be longer than for the others, because $|P| = |G|$ and $P_0 = G_0$; this means execution will progress into loop iteration $i + 1 = 1$, rather than early-abort at line 7 in iteration $i = 0$. As such, $G_0 = \texttt{'p'}$ is deemed correct in this guess, which we now fix. To recover P_i in iteration $i = 1$, we use a sequence of guesses:

$$G \in \langle \texttt{"paaaaaa"}, \texttt{"pbaaaaa"}, \texttt{"pcaaaaa"}, \ldots, \texttt{"pzaaaaa"} \rangle$$

during interaction with $\mathcal{T}$, that is, the ith character cycles through all possibilities, while the others remain fixed. The execution latency for guess $G = \texttt{"peaaaa"}$ will be longer than for the others, because $|P| = |G|$, $P_0 = G_0$ and $P_1 = G_1$; this means execution will progress into loop iteration $i + 1 = 2$, rather than early-abort at line 7 in iteration $i = 1$. As such, $G_1 = \texttt{'e'}$ is deemed correct in this guess, which we now fix. The attack continues in the same way for $0 \leq i < n$, until P is eventually fully recovered.

Since this is an explanatory attack, it is easy to assess it in terms of both efficacy and efficiency versus intuitive alternatives. For example, we could consider a brute-force attack: by essentially trying every n-character guess, this strategy assumes knowledge of n, requires $O(|A|^n)$ guesses and guarantees recovery of P. Or, we could consider a dictionary attack: by first constructing a dictionary of common (or likely) passwords $D = \{\texttt{"password"}, \texttt{"qwerty"}, \texttt{"admin"}, \ldots\}$ to use as guesses, this strategy does not assume knowledge of n, requires $O(|D|)$ guesses and does not guarantee recovery of P. In contrast, our side-channel attack does not assume knowledge of n, requires $O(|A| \cdot n)$ guesses and guarantees recovery of P; in a sense, the additional information allows it to offer greater efficiency than the brute-force attack *and* greater efficacy than the dictionary attack.

1.2.2. *Example 1.2: an attack on* `xtime`*-based AES*

1.2.2.1. *Scenario*

Assuming use of AES-128 throughout, consider the attack model depicted in Figure 1.2(b): an adversary $\mathcal{E}$ interacts with a target device $\mathcal{T}$, which stores some embedded, security-critical data k (an AES cipher key). When provided with input m (an AES plaintext), $\mathcal{T}$ first performs the computation detailed by the algorithm AES.ENC (i.e. AES encryption) and then responds with c (an AES ciphertext) as output. $\mathcal{E}$ can *also* measure the execution latency of computation performed by $\mathcal{T}$, so we attempted to exploit this information with the aim of recovering k.

1.2.2.2. *Analysis*

AES is an iterative block cipher based on an SP network; as one of three possible parameterizations, AES-128 uses a 16 byte cipher key and $b = 16$ byte block size. At a low level, it operates on elements in the finite field $\mathbb{F}_{2^8}$ realized as $\mathbb{F}_2[\mathbf{x}]/p(\mathbf{x})$, where $p(\mathbf{x}) = \mathbf{x}^8 + \mathbf{x}^4 + \mathbf{x}^3 + \mathbf{x} + 1$. A short-hand is often adopted for elements of this field, meaning, for example, $a(\mathbf{x}) = \mathbf{x}^4 + \mathbf{x} + 1 \equiv 13_{(16)}$, suggesting that each element can be represented by a byte (whose ith bit represents the ith coefficient). Elements of $\mathbb{F}_{2^8}$ are collected into (4×4)-element state and round key matrices; the ith row and jth column of such a matrix relating to round r is denoted $s_{i,j}^{(r)}$ and $rk_{i,j}^{(r)}$ respectively, with super- and/or subscripts omitted whenever irrelevant. The process of encryption initializes $s^{(0)} = m$ and $rk^{(0)} = k$, in both cases filling the left-hand matrix in a column-wise manner using content from the right-hand sequence of bytes.

$\mathcal{T}$ adopts an implementation strategy common in constrained platforms, targeting a balance that trades-off higher execution latency for lower memory footprint. This can be summarized as follows. First, it pre-computes the S-box: doing so implies it is realized using a table look-ups, rather than any computation. Second, it updates (or evolves) the cipher key, computing each round key during encryption rather than pre-computing them. Third, and finally, it uses the algorithm:

```
1 algorithm XTIME(a) begin
2 |   if a_7 = 1 then
3 |   |   return (a ≪ 1) ⊕ 11B_(16)
4 |   else
5 |   |   return (a ≪ 1)
6 |   end
7 end
```

to compute the multiplication-by-**x** operation, that is, $a(\mathbf{x}) \cdot \mathbf{x} \pmod{p(\mathbf{x})}$. On lines 3 and 5, the unconditional left-shift captures multiplication by **x**. On line 3, however,

the conditional XOR captures a reduction modulo $p(\mathbf{x})$; note that such a reduction is only required when $a_7 = 1$. Using this algorithm, it applies:

$$\text{MIX-COLUMN}\left(\begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix}\right) = \begin{bmatrix} 02_{(16)} & 03_{(16)} & 01_{(16)} & 01_{(16)} \\ 01_{(16)} & 02_{(16)} & 03_{(16)} & 01_{(16)} \\ 01_{(16)} & 01_{(16)} & 02_{(16)} & 03_{(16)} \\ 03_{(16)} & 01_{(16)} & 01_{(16)} & 02_{(16)} \end{bmatrix} \times \begin{bmatrix} s_{0,j} \\ s_{1,j} \\ s_{2,j} \\ s_{3,j} \end{bmatrix}$$

$$= \begin{bmatrix} \texttt{xtime}(s_{0,j}) \oplus (\texttt{xtime}(s_{1,j}) \oplus s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s_{0,j} \oplus \texttt{xtime}(s_{1,j}) \oplus (\texttt{xtime}(s_{2,j}) \oplus s_{2,j}) \oplus s_{3,j} \\ s_{0,j} \oplus s_{1,j} \oplus \texttt{xtime}(s_{2,j}) \oplus (\texttt{xtime}(s_{3,j}) \oplus s_{3,j}) \\ (\texttt{xtime}(s_{0,j}) \oplus s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus \texttt{xtime}(s_{3,j}) \end{bmatrix}$$

to each jth column of the state within MIXCOLUMNS.

In summary, the execution latency of AES encryption is data dependent: this stems from the implementation strategy adopted, whereby the execution latency of `xtime` and therefore MIX-COLUMNS is data dependent.

1.2.2.3. *Exploitation*

The attack strategy aims to recover k iteratively, that is, byte-by-byte. Let k_j and m_j denote the jth byte of plaintext and cipher key, respectively. In any tth iteration of the attack, we aim to recover the target byte k_t based on the observation that, in the first MIX-COLUMNS invocation, the computation:

$$\texttt{xtime}(\text{S-BOX}(m_t \oplus k_t))$$

is performed at least once; we term this the target `xtime`. Doing so involves the following steps, parameterized by N and M:

1) Compute the $(256 \times N)$-element matrix $\mathcal{H}$ such that:

$$\mathcal{H}_{i,j} = \begin{cases} 1 \text{ if S-BOX}(i \oplus j)_7 = 1 \\ 0 \text{ if S-BOX}(i \oplus j)_7 = 0 \end{cases}$$

Across the rows, each $0 \leq i < 256$ captures a possible value of k_t; across the columns, each $0 \leq j < N$ captures a possible value of m_t (meaning $N = 256$ will consider *all* possible values). As such, each $\mathcal{H}_{i,j}$ captures whether or not the target `xtime` performs a conditional XOR for specific k_t and m_t.

2) Compute the $(N \times M)$-element matrix:

$$\mathcal{S}_{i,j} = \{\langle s_0, s_1, \ldots, s_{b-1} \rangle \mid s_k = i \text{if } k = t,\ s_k \text{is random if } k \neq t\}$$

of plaintexts. So, since all plaintexts in the ith row have the same tth byte, they all yield the same input to the target `xtime`. Let $\mathcal{R}_{i,j}$ denote the execution latency arising from encryption of $\mathcal{S}_{i,j}$, which we measure via interaction with $\mathcal{T}$, and $\bar{\mathcal{R}}_i$ denote the mean of execution latencies across the ith row of $\mathcal{R}$.

The idea is that, provided N and M are large enough, $\bar{\mathcal{R}}_i$ will reflect the target `xtime`, that is, it will be larger or smaller depending on whether or not the conditional XOR occurs. Put another way, with some probability of error, $\bar{\mathcal{R}}_i$ indicates the value of S-BOX$(i \oplus k_t)_7$. To recover k_t, we compare each $\bar{\mathcal{R}}$ with each $\mathcal{H}_i$, for example, using the Pearson correlation coefficient: the i which yields the strongest correlation indicates the value of k_t.

1.2.3. *Example 1.3: an attack on Montgomery-based RSA*

1.2.3.1. *Scenario*

Consider the attack model depicted in Figure 1.2(c): an adversary $\mathcal{E}$ interacts with a target device $\mathcal{T}$, which stores the embedded, security-critical data (N, d) (an RSA private key). When provided with input c (an RSA ciphertext), $\mathcal{T}$ first performs the computation detailed by the algorithm RSA.DEC (i.e. RSA decryption) and then responds with m (an RSA plaintext) as output. $\mathcal{E}$ can *also* measure the execution latency of computation performed by $\mathcal{T}$, so attempts to exploit this information with the aim of recovering d.

1.2.3.2. *Analysis*

$\mathcal{T}$ uses the left-to-right, binary (or square-and-multiply) exponentiation algorithm. Note that all modular squaring and multiplication operations, for example, in lines 4 and 6, are based on the use of Montgomery reduction. This requires pre-computation of ω and ρ from N, such that $\hat{x} = x \cdot \rho \pmod N$ then denotes the Montgomery representations of some x.

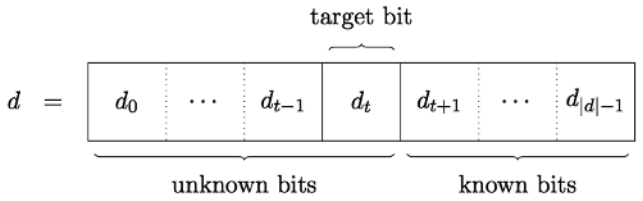
The execution latency of modular exponentiation is data dependent, because the execution of line 6 is conditional on d_i. An execution latency measurement may therefore allow us to infer how often line 6 is executed, and hence the Hamming weight of (or number of bits equal to 1 in) d. Beyond this, we need to analyze the MONTMUL algorithm as used to support the exponentiation algorithm. The steps involved are captured as follows:

```
1 algorithm MontMul(x, y, N) begin
2 |   r ← x · y
3 |   r ← (r + ((r · ω) mod ρ) · N)/ρ
4 |   if r >= N then r ← r − N
5 |   return r
6 end
```

where line 2 performs an integer multiplication, line 3 performs a Montgomery reduction, and then, finally, line 4 performs a conditional subtraction to fully reduce r modulo N. From this, we conclude that the execution latency of MONTMUL is also data dependent, because the execution of line 4 is dependent on both the operands x and y and the modulus N; as used to support exponentiation, this implies dependence on c (because it forms one of the operands), and, crucially, d (because it controls when, where and how MONTMUL is invoked).

1.2.3.3. *Exploitation*

The attack strategy aims to recover d iteratively, that is, bit-by-bit, working from the most-significant toward the least-significant bit. In any tth iteration of the attack, d is partially known and partially unknown, which we can depict as follows:



That is, we already know some more-significant bits of d (noting that it must be the case that some $d_{|d|-1} = 1$; otherwise $d = 0$) and so aim to recover the next less significant, as yet unknown target bit d_t. Let $c[k]$ denote some kth ciphertext, randomly selected modulo N for $0 \leq k < n$; for each such $c[k]$, we use (partial) knowledge of d to simulate a (partial) decryption as would be performed by $\mathcal{T}$. Doing so involves the following steps:

1) Start by setting $c = c[k]$, then simulating lines 2 and 3, that is, computing $\hat{c} \leftarrow \text{MONTMUL}(c, \rho^2, N)$ and $\hat{r} \leftarrow \text{MONTMUL}(1, \rho^2, N)$.

2) Step the simulation forward until we reach the start of loop iteration t (between lines 4 and 5). We can do so because d_i is known for $t < i < |d|$, meaning the simulation can perform exactly the same computation that $\mathcal{T}$ would.

3) Step the simulation forward until we reach the end of loop iteration t (between lines 8 and 9). We can do so because although d_i is unknown for $i = t$, we can fork the simulation to reflect guessing both possibilities:

a) for the simulation fork that guesses $d_i = 0$, line 6 is not executed so we do not update $\hat{r}$;

b) for the simulation fork that guesses $d_i = 1$, line 6 is executed so we update $\hat{r} \leftarrow \text{MONTMUL}(\hat{r}, \hat{c}, N)$.

4) Step the simulation forward until we reach the middle of loop iteration $t + 1$ (between lines 5 and 6). That is, we update $\hat{r} \leftarrow \text{MONTMUL}(\hat{r}, \hat{r}, N)$ and, in doing

so, record whether or not the conditional subtraction, that is, line 4 of MONTMUL, is executed.

5) Classify c by placing it into set:

$\mathcal{S}_0$ if the simulation fork guessed $d_i = 0$ and no conditional subtraction is executed
$\mathcal{S}_1$ if the simulation fork guessed $d_i = 0$ and conditional subtraction is executed
$\mathcal{S}_2$ if the simulation fork guessed $d_i = 1$ and no conditional subtraction is executed
$\mathcal{S}_3$ if the simulation fork guessed $d_i = 1$ and conditional subtraction is executed

This means c is placed into exactly *two* sets: either $\mathcal{S}_0$ or $\mathcal{S}_1$ arising from the simulation fork that guessed $d_i = 0$, and either $\mathcal{S}_2$ or $\mathcal{S}_3$ arising from the simulation fork that guessed $d_i = 1$.

The idea is that we have intentionally constructed the sets so there should be a difference between the execution latency for the ciphertexts in $\mathcal{S}_0$ versus those in $\mathcal{S}_1$, and the ciphertexts in $\mathcal{S}_2$ versus those in $\mathcal{S}_3$. Under the assumption that conditional subtractions will occur more or less at random during exponentiation, ciphertexts in the latter will, on average, execute an extra conditional subtraction versus those in the former. However, only one of the guesses made by the simulation is actually correct, so that difference should only be evident in either the first or second case. Put another way, for the correct (respectively, incorrect) guess, the simulated computation will (respectively, will not) match the actual computation by $\mathcal{T}$; therefore, for the correct (respectively, incorrect) guess, the number of conditional subtractions that occur during exponentiation of ciphertexts in one set will differ from (respectively, be approximately equal to) those in the other set. So, let

$$\Lambda(\mathcal{S}_k) = \frac{1}{|\mathcal{S}_k|} \sum_{c \in \mathcal{S}_k} \Lambda(c)$$

denote the average execution latency of ciphertexts in set $\mathcal{S}_k$, where $\Lambda(c)$ denotes the execution latency for some ciphertext c: we measure this by interacting with $\mathcal{T}$. One of three cases will apply:

$$\begin{array}{lcl} \mathrm{abs}(\ \Lambda(\mathcal{S}_0) - \Lambda(\mathcal{S}_1)\ ) > \mathrm{abs}(\ \Lambda(\mathcal{S}_2) - \Lambda(\mathcal{S}_3)\ ) & \Rightarrow & d_t = 0 \\ \mathrm{abs}(\ \Lambda(\mathcal{S}_0) - \Lambda(\mathcal{S}_1)\ ) < \mathrm{abs}(\ \Lambda(\mathcal{S}_2) - \Lambda(\mathcal{S}_3)\ ) & \Rightarrow & d_t = 1 \\ \mathrm{abs}(\ \Lambda(\mathcal{S}_0) - \Lambda(\mathcal{S}_1)\ ) \simeq \mathrm{abs}(\ \Lambda(\mathcal{S}_2) - \Lambda(\mathcal{S}_3)\ ) & & \end{array}$$

In the first (top) and second (middle) cases, the difference we expect will allow us to infer the value of d_t. In the third (bottom) case, however, the difference is too close to allow confident inference of d_t; this may be due to, for example, use of too small an n or the impact of noise.

Having recovered d_t, we proceed with the next, $(t+1)$th attack iteration and recover the next, $(t+1)$th bit of d in the same way; this continues until d is eventually fully recovered.

1.2.4. *Example 1.4: a padding oracle attack on AES-CBC*

1.2.4.1. *Scenario*

Assuming use of AES-128 throughout, consider the attack model depicted in Figure 1.2(d): an adversary $\mathcal{E}$ interacts with a target device $\mathcal{T}$, which stores the embedded, security-critical data k (some key material). When provided with input (iv, c), $\mathcal{T}$ first performs the computation detailed by the algorithm PROCESS, then responds with r as output. $\mathcal{E}$ obtains a ciphertext (iv^*, c^*), which is the encryption of some unknown plaintext m^* under k. It can *also* measure the execution latency of computation performed by $\mathcal{T}$, so attempts to exploit this information with the aim of recovering m^*.

1.2.4.2. *Analysis*

Let $x[i]_j$ denote the jth byte within the ith block of some n-block plaintext or ciphertext x; this means $0 \leq i < n$ and $0 \leq j < b$, assuming a b-byte block size.

At a high level, PROCESS can be described line-by-line as follows. Line 2 decrypts the ciphertext c, parsing the result m' as a plaintext m, some padding ρ and a tag τ. Then, line 3 checks whether ρ is valid, aborting if not; line 4 checks whether τ is valid, aborting if not; and, finally, line 5 processes m using some function f to produce r. At a low level, some further details are important: note that line 2 decrypts c using AES in Cipher Block Chaining (CBC) mode using the key k_1, and line 3 assumes τ is a SHA1-based HMAC Message Authentication Code (MAC) tag and checks validity accordingly using the key k_2. The padding scheme used to specify ρ is also important, but not detailed within the description of PROCESS. Although alternatives exist, assume a TLS-like scheme is used: let

$$\rho(\alpha) \;=\; \langle \underbrace{\alpha,\ \alpha,\ \ldots,\ \alpha}_{\alpha\text{byte(s)}},\ \alpha \rangle$$

denote a valid padding sequence, which contains α byte(s) equal to α and then 1 compulsory byte equal to α, which records the total padding length. Note that if $m \parallel \tau$ is l bytes long, then $\rho = \rho(\, b - (l+1) \bmod b \,)$.

The execution latency of PROCESS is data dependent: if t_3 denotes the execution latency when $\mathcal{T}$ early-aborts on line 3 based on ρ, t_4 denotes the execution latency when $\mathcal{T}$ early-aborts on line 4 based on τ, and t_5 denotes the execution latency when $\mathcal{T}$ executes line 5, then, irrespective of their concrete values, $t_3 < t_4 < t_5$. In fact,

we can focus on the ability to distinguish between these cases by defining a so-called padding oracle: for the scenario at hand, this would be something like:

$$\mathcal{O}(iv, c) = \begin{cases} \textbf{true} \text{ if } \mathsf{AES\text{-}CBC}.\textsc{Dec}(k_1, iv, c) \text{has} \quad \text{valid padding} \\ \textbf{false} \text{ if } \mathsf{AES\text{-}CBC}.\textsc{Dec}(k_1, iv, c) \text{has invalid padding} \end{cases}$$

Doing so can be viewed as an abstraction of the underlying mechanism used to measure execution latency, thus separating it from the attack strategy.

1.2.4.3. *Exploitation*

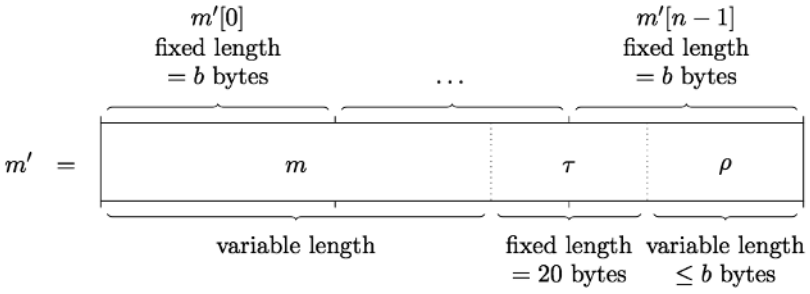
Use of AES in CBC mode means that $\mathcal{T}$ will compute:

$$m'[i] = \mathsf{AES}.\textsc{Dec}(k_1, c[i]) \oplus x_i$$

where

$$x_i = \begin{cases} iv & \text{if } i = 0 \\ c[i-1] & \text{if } i > 0 \end{cases}$$

for $0 \leq i < n$. The resulting plaintext m' is then parsed into the fields m, τ, and ρ, which we can depict as follows:



From this specific example, we can infer several more general facts. First, ρ will occur in $m'[n-1]$, that is, the last, $(n-1)$th block of m'. Second, although $|\rho|$ is unknown, we *do* know $|\rho| \leq b$ because the point of including it at all is to ensure $|m'| = 0 \pmod{b}$. Third, the definition of CBC mode means that:

$$m'[n-1] = \mathsf{AES}.\textsc{Dec}(k_1, c[n-1]) \oplus x_i.$$

As such, by controlling x_i (meaning either iv or $c[n-2]$, depending on i), we can control $m'[n-1]$ and hence also the ρ then checked for validity by $\mathcal{T}$. For example,

consider a two-block ciphertext c. If we construct an input (iv, c') where $c' = (c[0] \oplus \delta) \ \| \ c[1]$, for some δ, that is, $c'[0] = c[0] \oplus \delta$ and $c'[1] = c[1]$, then $\mathcal{T}$ will compute:

$$\begin{aligned} m'[0] &= \mathsf{AES}.\textsc{Dec}(k_1, c'[0]) \oplus iv \\ &= \mathsf{AES}.\textsc{Dec}(k_1, c[0] \oplus \delta) \oplus iv \\ m'[1] &= \mathsf{AES}.\textsc{Dec}(k_1, c'[1]) \oplus c'[0] \oplus \delta \\ &= \mathsf{AES}.\textsc{Dec}(k_1, c[1]) \oplus c[0] \oplus \delta \end{aligned}$$

That is, $\mathcal{T}$ will compute an m' where $m'[0]$ is randomized by and $m'[1]$ is controlled by the δ selected; since $m'[1]$ is the $(n-1)$th block in m', δ will also control ρ and hence the padding oracle result, that is,

$$\mathcal{O}(iv, (c[0] \oplus \delta) \ \| \ c[1]).$$

Based on these facts, and assuming again that $|c^*| = 2$, the attack proceeds in two phases:

1) In this phase, the attack strategy constructs a so-called "recover last byte(s)" oracle. First, select a random b-byte block δ and search for a value of δ_{b-1} such that:

$$\mathcal{O}(iv, (c^*[0] \oplus \delta) \ \| \ c^*[1]) = \textbf{true}.$$

Since the padding of m' is deemed to be valid, we know that one of:

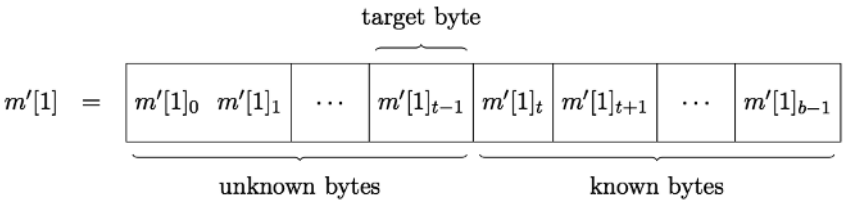
$$\begin{array}{lcll} m'[1]_{b-1} & = & m^*[1]_{b-1} \oplus \delta_{b-1} & = \ \rho(0) \\ m'[1]_{b-2,\ldots,b-1} & = & m^*[1]_{b-2,\ldots,b-1} \oplus \delta_{b-2,\ldots,b-1} & = \ \rho(1) \\ m'[1]_{b-3,\ldots,b-1} & = & m^*[1]_{b-3,\ldots,b-1} \oplus \delta_{b-3,\ldots,b-1} & = \ \rho(2) \\ & \vdots & & \vdots \end{array}$$

is true: however, we do not know *which* one. So, to recover one (or more) byte(s) of m', we need $|\rho|$, that is, the padding length. To recover it, start from $t = 0$ and alter (e.g. increment) δ_t. If

$$\mathcal{O}(iv, (c^*[0] \oplus \delta) \ \| \ c^*[1]) = \textbf{false},$$

then t marks the initial padding byte (because we just altered it, thus invalidating the padding) and implies that $|\rho| = b - t$; otherwise, increment t and try again.

2) In this phase, the attack strategy constructs a so-called "block decryption" oracle. As mentioned above, we have already recovered the most-significant $b - t$ bytes of m', that is, the *natural* padding: we therefore have:



for some t where

$$m'[1]_{t,\ldots,b-1} \;=\; m^*[1]_{t,\ldots,b-1} \oplus \delta_{t,\ldots,b-1} \;=\; \rho(b-t-1).$$

To recover the next least-significant byte of $m'[1]$, we try to control δ so an *extended* padding is produced. To do so, we begin by incrementing the known padding bytes by setting $\delta_j = m'[1]_j \oplus (b-t)$ for $t \leq j < b$, and then we apply a similar strategy as above by searching for a value of δ_{t-1} such that the padding of m' is deemed valid: this recovers $m'[1]_{t-1}$. Then, we can iterate for each next least-significant byte to recover the whole of $m'[1]$ (and hence $m^*[1]$).

Of course, as described, this process is specific to recovery of $m'[1]$ and, even then, the case where $|c^*| = 2$. However, with minor alterations the same underlying concept generalizes to cater for recovery of any $m'[i]$ and for any $|c^*|$.

1.3. Example mitigations

Fundamentally, mitigating side-channel attacks based on execution latency can be achieved by preventing said execution latency being data dependent. Although true, this vastly understates the challenge in doing so, which is evidenced by a variety of points. For example, first, the basic properties of scale and complexity can be hugely challenging. On the one hand, there is a large design space of techniques to consider; on the other hand, they often need to be applied to and so integrated with a large, complex software code base that may be constrained, for example, with respect to requirements of some overarching protocol. Second, the underlying issue is evident in, and so potentially must be addressed at various levels of abstraction, for example, both at a high level, relating to the design algorithms, *and* at a low level, relating to their implementation, and, ultimately execution. Third, security may be *a* metric, not *the* metric of interest; there can be pressure to compromise on security in order to meet efficiency targets, for example, which demands carefully considered trade-offs.

Although, there is seldom a panacea for such challenges, (1) clear understanding of the context, constraints, problem and platform, and (2) well-informed selection and application of implementation techniques are centrally important in producing an outcome, which is fit for purpose. We can find a detailed treatment of relevant topics in Chapters 7 of Volume 2. For completeness, however, the following provides

an illustrative overview of pertinent examples, framed around phases of a typical development cycle (implying iteration over said phases may be required).

1) *Design*: at a high(er) more abstract level, the first phase specifies a design for what will be implemented. Appropriate high-level choices will often have the largest impact from an efficiency perspective, and the same is true from a security perspective: care in this phase can make subsequent phases significantly easier.

Among many options, we might consider algorithm selection (in the context of ECC, e.g. use of the Montgomery ladder and/or complete addition formula), parameter selection (in the context of RSA, e.g. of ρ to remove conditional subtraction in Montgomery multiplication), techniques related to alternative representation of data (e.g. bit-slicing), techniques related to alternative computational models (e.g. based on URISC), approaches based on blinding (e.g. of an RSA private exponent), or approaches based on padding (e.g. to worst-case execution latency).

2) *Implementation*: at a low(er), more concrete level, the next phase implements what has been specified. Care with respect to how a high(er) level implementation is translated into a low-level instruction sequence by a tool chain, *and* how that instruction sequence is executed by the platform, is crucial. For example, note that many compiler optimizations intended to have a positive impact on efficiency *may* have a negative impact on security, e.g. by "un-doing" features intended to ensure data-independent execution latency.

Among many options, we might consider approaches based on control-flow linearization (or flattening) to yield straight-line code (e.g. software multiplexer gadgets, or via conditional-move instructions), approaches based on control-flow balancing, approaches that yield constant memory access patterns (e.g. within window-based modular exponentiation) and approaches that favor computation versus memory access (e.g. use of AES-NI versus look-ups into pre-computed S-box or T-table content).

3) *Evaluation*: in the final phase, we aim to evaluate what has been implemented, and, ideally, gather evidence and hence confidence that the associated execution latency is data independent.

Among many options, we might consider approaches based on (static) verification (including use of baseline assumptions, e.g. as provided by RISC-V `Zkt` pseudo-extension) and/or on (dynamic) experimentation (e.g. `ctgrind` and `dudec`).

1.4. Notes and further references

– Section 1.1. There are many good side-channel attack overviews. Anderson (2020) offers a accessible, general example with a range of historical context, while Koeune (2011) covers attacks based on execution latency specifically. The first

publication related to such attacks was by Kocher (1996). Nearly a decade later, in Brumley and Boneh (2003) then expanded the scope by demonstrating remote applicability, that is, over a network. The same general concept has been further explored within numerous different scenarios, for example, by Schindler (2000), Aciiçmez et al. (2005) and Brumley and Tuveri (2011). The work of Kocher is seminal, which presents both specific attacks, for example, on RSA and DSA, *and* articulates the more general threat; we can make a strong argument that it represents the foundation of modern research on side-channel attacks. For example, it already notes that "*RAM cache hits can produce timing characteristics in implementations*" and thus identifies the more general challenge of data-dependent execution latency arising from the platform; the range of components studied along similar lines is vast, for example, including behavior of integer, such as Großschädl et al. (2010) and floating-point, such as Andrysco et al. (2015) and Kohlbrenner and Shacham (2017), instruction execution, plus wider, system-level components such as the C compiler (Kaufmann et al. 2016), the JIT compiler (Brennan et al. 2020) and the garbage collector (Pedersen and Askarov 2017).

Note that this chapter intentionally avoids coverage of so-called micro-architectural attacks, many of which are based on data-dependent execution latency in some way: Chapter 2 offers detailed coverage, but, for example, various related books such as Rebeiro et al. (2015) and surveys such as Ge et al. (2018) and Szefer (2019) may also be interesting. Also, although we adopted a focus on cryptographic examples, it is vital to stress the applicability of the same underlying concepts in other contexts: examples include recovery of key-stroke information (Song et al. 2001; Tey et al. 2013; Lipp et al. 2017; Balagani et al. 2018), recovery of resources associated with web-browsing (Felten and Schneider 2000; Kotcher et al. 2013; van Goethem et al. 2015), and traffic analysis (Murdoch and Danezis 2005; Wang et al. 2005). Likewise, it may be interesting to consider that data-dependent execution latency has *constructive* as well as *destructive* use-cases: examples include IP protection (Donda et al. 2015).

– Section 1.2.1. Among numerous explanatory examples, of side-channel attacks in general and those specifically based on execution latency, the presentation by Naccache and Tunstall (2000) is an often-cited instance. Although not strictly comparable to MATCH, it is interesting to note that the manual page for the C `memcmp` function warns "*[d]o not use `memcmp()` to compare security critical data, such as cryptographic secrets, because the required CPU time depends on the number of equal bytes"*, and so hints at similar real-world behavior.

– Section 1.2.2. The attack description follows the study that of Koeune and Quisquater (1999), which predates the standardization of Advanced Encryption Standard (AES); although compatible in the scenario used, the authors consider then AES *candidate* Rijndael.

– Section 1.2.3. Rivest et al. (1978) introduced what we now know as the RSA public-key encryption scheme. This attack demands careful understanding of the associated implementation. As such, Gordon (1998) offers a general overview of exponentiation algorithms; Montgomery (1985) describes the eponymous Montgomery reduction technique, with Koç et al. (1996) surveying various implementation options for it. The attack strategy was already outlined by Kocher (1996), with further analysis later offered by Dhem et al. (1998). The attack description follows the latter, focusing on the second case they consider in Dhem et al. (1998), section 3.4, that is, attacking the modular squaring. Note that similar attacks are possible for other scenarios: examples include Mittmann and Schindler (2021), who consider Barrett (1987) reduction, and Bakhshi and Sadeghiyan (2007), who consider Blakley (1983) reduction.

– Section 1.2.4. This attack demands careful understanding of the associated implementation. As such, note that the components involved are all standardized in some way: see, for example, AES (2001), CBC mode (National Institute of Standards and Technology (NIST) 2001b, section 6.2), HMAC (2008) and the TLS-like padding scheme assumed (Dierks and Rescorla 2008, section 6.2.3.2). Set within a symmetric-focused scenario, the attack description follows that of Vaudenay (2002) (later generalized by Black and Urtubia 2002). A wide range of work has applied this approach to real-world targets: examples include Canvel et al. (2003) and Paterson and Yau (2004) and Yau et al. (2005), who consider it as applied to TLS- and ISO/IEC-based cases, respectively, Avoine and Ferreira (2018), who consider padding oracle attacks on a SCP02-compliant smart-card, and both Lucky 13 (AlFardan and Paterson 2013) and Lucky Microseconds (Albrecht and Paterson 2016), which represent further attacks on CBC-based encryption in (D)TLS. It is also important to acknowledge conceptually analogous alternatives set within an asymmetric-focused scenario: important examples include the attacks due to Bleichenbacher (1998) (see also Jager et al. (2012); Meyer et al. (2014); Böck et al. (2018); Ronen et al. (2019)) and later Manger (2001) (see also Strenzke (2010)).

– Section 1.3. Coverage of mitigation, or countermeasures, is challenging in the sense the topic is broad and diverse. From a software perspective only, in order to reduce the scope somewhat, an accessible starting point would be the "coding rules" for implementations maintained by Aumasson Github (n.d.a) and others. At a high level, data-dependent control-flow is supported by numerous, context-dependent approaches. For RSA, relevant examples include various work, for example, Walter (1999) and Hachez and Quisquater (2000), which considers elimination the data-dependent conditional subtraction in Montgomery multiplication. For ECC, relevant examples include use of complete formula, for example, as per Renes et al. (2016), and regular scalar multiplication based on the Montgomery ladder (Montgomery 1987) (thoroughly investigated by Joye and Yen (2003)); Joye (2004) and Oswald (2004) offer a general overview from destructive and constructive perspectives. At a low level, Molnar et al. (2006) propose the program counter

security model, which is essentially a way to formally reason about data-dependent control-flow; such reasoning includes both checking for data-(in)dependence, and transformation of instruction sequences. Coppens et al. (2009) cover a range of related techniques for, for example, control-flow linearization, automation within an LLVM-based compiler back-end for x86 platforms.

It seems fair to say that, at least at the time of writing, tooling which supports (semi-)automatic analysis of implementations is limited but improving. As part of a wider survey, Barbosa et al. (2021), section IV present an overview; relevant examples can be categorized into static, for example, ct-verif (Almeida et al. 2016), FlowTracker (Rodrigues et al. 2016), MicroWalk (Wichelmann et al. 2018), SideTrail (Athanasiou et al. 2018) and FaCT (Cauligi et al. 2019) and dynamic, for example, `dudec` (Reparaz et al. 2017), `ctgrind` (Github n.d.b), Triggerflow (Gridin and Brumley 2019) and DATA (Weiser et al. 2018, 2020) cases.

1.5. References

Aciiçmez, O., Schindler, W., Koç, Ç. (2005). Improving Brumley and Boneh timing attack on unprotected SSL implementations. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, 139–146.

Albrecht, M.R. and Paterson, K.G. (2016). Lucky microseconds: A timing attack on amazon's s2n implementation of TLS. In *35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Vienna, 622–643.

AlFardan, N.J. and Paterson, K.G. (2013). Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, Berkeley, CA, 526–540.

Almeida, J.B., Barbosa, M., Barthe, G., Dupressoir, F., Emmi, M. (2016). Verifying constant-time implementations. In *Proceedings of the 25th USENIX Security Symposium*, Austin, 53–70.

Anderson, R. (2020). Side channels. In *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3rd edition. Anderson, R. (edition). John Wiley & Sons, New York. doi: 10.5555/1373319.

Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., Shacham, H. (2015). On subnormal floating point and abnormal timing. In *IEEE Symposium on Security and Privacy*, 623–639.

Athanasiou, K., Cook, B., Emmi, M., MacCarthaigh, C., Schwartz-Narbonne, D., Tasiran, S. (2018). SideTrail: Verifying time-balancing of cryptosystems. In *Verified Software. Theories, Tools and Experiments (VSTTE)*. Springer, Cham. doi: 10.1007/978-3-030-03592-1_12.

Avoine, G. and Ferreira, L. (2018). Attacking GlobalPlatform SCP02-compliant smart cards using a padding oracle attack. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2), 149–170 [Online]. Available at: https://tches.iacr.org/index.php/TCHES/article/view/878.

Bakhshi, B. and Sadeghiyan, B. (2007). A timing attack on Blakley's modular multiplication algorithm, and applications to DSA. *Applied Cryptography and Network Security*, 129–140.

Balagani, K.S., Conti, M., Gasti, P., Georgiev, M., Gurtler, T., Lain, D., Miller, C., Molas, K., Samarin, N., Saraci, E. et al. (2018). SILK-TV: Secret information leakage from keystroke timing videos. In *European Symposium on Research in Computer Security*, 263–280.

Barbosa, M., Barthe, G., Bhargavan, K., Blanchet, B., Cremers, C., Liao, K., Parno, B. (2021). SoK: Computer-aided cryptography. In *IEEE Symposium on Security and Privacy*, San Fransisco. doi: 10.1109/SP40001.2021.00008.

Barrett, P. (1987). Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology*, 311–323.

Black, J. and Urtubia, H. (2002). Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In *Proceedings of the 11th USENIX Security Symposium*, 327–338.

Blake, I., Seroussi, G., Smart, N. (eds) (2004). *Advances in Elliptic Curve Cryptography*, 1st edition. Cambridge University Press, Cambridge. doi: 10.1017/CBO9780511546570.

Blakley, G. (1983). A computer algorithm for calculating the product AB modulo M. *IEEE Transactions on Computers*, 32(5), 497–500. doi: 10.1109/TC.1983.1676262.

Bleichenbacher, D. (1998). Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO '98*, 1–12.

Böck, H., Somorovsky, J., Young, C. (2018). Return of Bleichenbacher's oracle threat (ROBOT). In *27th Usenix Security Symposium*, 817–849.

Brennan, T., Rosner, N., Bultan, T. (2020). JIT leaks: Inducing timing side channels through just-in-time compilation. In *Proceedings - 2020 IEEE Symposium on Security and Privacy*, 1207–1222.

Brumley, D. and Boneh, D. (2003). Remote timing attacks are practical. In *12th Usenix Security Symposium*.

Brumley, B.B. and Tuveri, N. (2011). Remote timing attacks are still practical. In *European Symposium on Research in Computer Security*, 355–371.

Canvel, B., Hiltgen, A.P., Vaudenay, S., Vuagnoux, M. (2003). Password interception in a SSL/TLS channel. In *Annual International Cryptology Conference*, 583–599.

Cauligi, S., Soeller, G., Johannesmeyer, B., Brown, F., Wahby, R., Renner, J., Grégoire, B., Barthe, G., Jhala, R., Stefan, D. (2019). FaCT: A DSL for timing-sensitive computation. In *Programming Language Design and Implementation (PLDI)*, June 22–26. ACM, Phoenix. doi: 10.1145/3314221.3314605.

Coppens, B., Verbauwhede, I., De Bosschere, K., De Sutter, B. (2009). Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the IEEE Symposium on Security and Privacy*, 45–60.

Dhem, J.-F., Koeune, F., Leroux, P.-A., Mestré, P., Quisquater, J.-J., Willems, J.-L. (1998). A practical implementation of the timing attack. In *Smart Card Research and Applications (CARDIS)*. Springer, Berlin, Heidelberg. doi: 10.1007/10721064_15.

Dierks, T. and Rescorla, E. (2008). The Transport Layer Security (TLS) protocol version 1.2. Internet Engineering Task Force (IETF) Request for Comments (RFC) 5246 [Online]. Available at: http://tools.ietf.org/html/rfc5246.

Donda, A.-T., Samarin, P., Samotyja, J., Lemke-Rust, K., Paar, C. (2015). Remote IP protection using timing channels. In *International Conference on Information Security and Cryptology*,

222–237.

Felten, E.W. and Schneider, M.A. (2000). Timing attacks on web privacy. In *Proceedings of the ACM Conference on Computer and Communications Security*, 25–32.

Ge, Q., Yarom, Y., Cock, D., Heiser, G. (2018).. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering (JCEN)*, 8, 1–27. doi: 10.1007/s13389-016-0141-6.

Github (n.d.a). Cryptocoding [Online]. Available at: https://github.com/veorq/cryptocoding.

Github (n.d.b). Checking that functions are constant time with Valgrind [Online]. Available at: https://github.com/agl/ctgrind.

van Goethem, T., Joosen, W., Nikiforakis, N. (2015). The clock is still ticking: Timing attacks in the modern web. In *CCS '15: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 1382–1393.

Gordon, D. (1998). A survey of fast exponentiation methods. *Journal of Algorithms*, 27, 129–146. doi: 10.1006/jagm.1997.0913.

Gridin, I. and Brumley, C.G.N.T.B. (2019). Triggerflow: Regression testing by advanced execution path inspection. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, Cham. doi: 10.1007/978-3-030-22038-9_16.

Großschädl, J., Oswald, E., Page, D., Tunstall, M. (2010). Side-channel analysis of cryptographic software via early-terminating multiplications. In *International Conference on Information Security and Cryptology*, 176–192.

Hachez, G. and Quisquater, J.-J. (2000). Montgomery exponentiation with no final subtractions: Improved results. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 293–301.

Jager, T., Schinzel, S., Somorovsky, J. (2012). Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. In *European Symposium on Research in Computer Security*, 752–769.

Joye, M. (2004). Defences against side-channel analysis. In *Advances in Elliptic Curve Cryptography*. Cambridge University Press, Cambridge. doi: 10.1017/CBO9780511546570.

Joye, M. and Yen, S.-M. (2003). The Montgomery powering ladder. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 291–302.

Kaufmann, T., Pelletier, H., Vaudenay, S., Villegas, K. (2016). When constant-time source yields variable-time binary: Exploiting Curve25519-donna built with MSVC 2015. In *Cryptology and Network Security: 15th International Conference*, 573–582.

Koç, Ç.K., Acar, T., Kaliski, B. (1996). Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3), 26–33. doi: 10.1109/40.502403.

Kocher, P.C. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, 104–113.

Koeune, F. (2011). *Timing Attack*, 2nd edition. Springer, New York. doi: 10.1007/978-1-4419-5906-5.

Koeune, F. and Quisquater, J.-J. (1999). A timing attack against Rijndael. Technical Report CG-1999/1, Université catholique de Louvain.

Kohlbrenner, D. and Shacham, H. (2017). On the effectiveness of mitigations against floating-point timing channels. In *26th Usenix Security Symposium*, 69–81.

Kotcher, R., Pei, Y., Jumde, P., Jackson, C. (2013). Cross-origin pixel stealing: Timing attacks using CSS filters. In *CCS '13: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 1055–1062.

Lipp, M., Gruss, D., Schwarz, M., Bidner, D., Maurice, C., Mangard, S. (2017). Practical keystroke timing attacks in sandboxed JavaScript. In *22nd European Symposium on Research in Computer Security, Proceedings*. Springer-Verlag, Italy, 191–209.

Manger, J. (2001). A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In *Annual International Cryptology Conference*, 230–238.

Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., Tews, E. (2014). Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks. In *Proceedings of the 23rd USENIX Security Symposium*, San Diego, 733–748.

Mittmann, J. and Schindler, W. (2021). Timing attacks and local timing attacks against Barrett's modular multiplication algorithm. *Journal of Cryptographic Engineering (JCEN)*, 11(4), 369–397. doi: 10.1007/s13389-020-00254-3.

Molnar, D., Piotrowski, M., Schultz, D., Wagner, D. (2006). The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*, 156–168.

Montgomery, P. (1985). Modular multiplication without trial division. *Mathematics of Computation*, 44(170), 519–521. doi: 10.1090/S0025-5718-1985-0777282-X.

Montgomery, P. (1987). Speeding the pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177), 243–264. doi: 10.1090/S0025-5718-1987-0866113-7.

Murdoch, S.J. and Danezis, G. (2005). Low-cost traffic analysis of tor. In *2005 IEEE Symposium on Security and Privacy*, 183–195.

Naccache, D. and Tunstall, M. (2000). How to explain side-channel leakage to your kids (invited talk). In *CHES 2000*, 229–230.

National Institute of Standards and Technology (NIST) (2001a). Advanced Encryption Standard (AES). Federal Information Processing Standard (FIPS) 197 [Online]. Available at: http://csrc.nist.gov.

National Institute of Standards and Technology (NIST) (2001b). Recommendation for Block Cipher Modes of Operation: Methods and Techniques. Special Publication 800-38A [Online]. Available at: http://csrc.nist.gov.

National Institute of Standards and Technology (NIST) (2008). The Keyed-Hash Message Authentication Code (HMAC). Federal Information Processing Standard (FIPS) 198 [Online]. Available at: http://csrc.nist.gov.

Oswald, E. (2004). Side-channel analysis. In *Advances in Elliptic Curve Cryptography*, 1st edition. Cambridge University Press, Cambridge. doi: 10.1017/CBO9780511546570.

Paterson, K.G. and Yau, A. (2004). Padding oracle attacks on the ISO CBC mode encryption standard. In *Cryptographers' Track at the RSA Conference*, 305–323.

Pedersen, M.V. and Askarov, A. (2017). From trash to treasure: Timing-sensitive garbage collection. In *2017 IEEE Symposium on Security and Privacy, SP 2017 – Proceedings*, 693–709.

Rebeiro, C., Mukhopadhyay, D., Bhattacharya, S. (2015). *Timing Channels in Cryptography*. Springer, Cham. doi: 10.1007/978-3-319-12370-7.

Renes, J., Costello, C., Batina, L. (2016). Complete addition formulas for prime order elliptic curves. In *Proceedings, Part I, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016*, 403–428.

Reparaz, O., Balasch, J., Verbauwhede, I. (2017). Dude, is my code constant time? In *Design, Automation & Test in Europe (DATE)*. IEEE, Lausanne. doi: 10.23919/DATE.2017.7927267.

Rivest, R.L., Shamir, A., Adleman, L.M. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 120–126.

Rodrigues, B., Fernando, M., Aranha, D. (2016). Sparse representation of implicit flows with applications to side-channel detection. In *International Conference on Compiler Construction (CC)*, ACM, New York, 110–120. doi: 10.1145/2892208.2892230.

Ronen, E., Gillham, R., Genkin, D., Shamir, A., Wong, D., Yarom, Y. (2019). The 9 lives of Bleichenbacher's CAT: New cache ATtacks on TLS implementations. In *2019 IEEE Symposium on Security and Privacy (SP)*, 435–452.

Schindler, W. (2000). A timing attack against RSA with the Chinese remainder theorem. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 109–124.

Song, D.X., Wagner, D.A., Tian, X. (2001). Timing analysis of keystrokes and timing attacks on SSH. In *10th Usenix Security Symposium*, Washington, D.C.

Strenzke, F. (2010). Manger's attack revisited. In *International Conference on Information and Communications Security*, 31–45.

Szefer, J. (2019). Survey of microarchitectural side and covert channels, attacks, and defences. *Journal of Hardware and Systems Security*, 3(3), 219–234. doi: 10.1007/s41635-018-0046-1.

Tey, C.M., Gupta, P., Gao, D., Zhang, Y. (2013). Keystroke timing analysis of on-the-fly web apps. In *International Conference on Applied Cryptography and Network Security*, 405–413.

Vaudenay, S. (2002). Security flaws induced by CBC padding – Applications to SSL, IPSEC, WTLS... In *EUROCRYPT '02: Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, 534–546.

Walter, C. (1999). Montgomery exponentiation needs no final subtractions. *IEE Electronics Letters*, 35(21), 1831–1832. doi: 10.1049/el:19991230.

Wang, X., Chen, S., Jajodia, S. (2005). Tracking anonymous peer-to-peer VoIP calls on the Internet. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, 81–91.

Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., Sigl, G. (2018). DATA – differential address trace analysis: Finding address-based side-channels in binaries. In *27th Usenix Security Symposium*, 603–620.

Weiser, S., Schrammel, D., Bodner, L., Spreitzer, R. (2020). Big numbers – Big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations. In *29th Usenix Security Symposium*, 1767–1784.

Wichelmann, J., Moghimi, A., Eisenbarth, T., Sunar, B. (2018). MicroWalk: A framework for finding side channels in binaries. In *Annual Computer Security Applications Conference (ACSAC)*. doi: 10.1145/3274694.3274741.

Yau, A.K.L., Paterson, K.G., Mitchell, C.J. (2005). Padding oracle attacks on CBC-mode encryption with secret and random IVs. In *International Workshop on Fast Software Encryption*, 299–319.