# What is Computer Science?

## An Information Security Perspective

Daniel Page ⟨dan@phoo.org⟩ and Nigel P. Smart ⟨csnps@bristol.ac.uk⟩

git # b4055dd3 @ 2019-05-13

# IS THE KEY TO YOUR CASH BEING LEAKED BY YOUR CACHE?

If you talk to people who design computers, they might curse what is often called the **memory wall**. The problem is that memory is quite slow in comparison to everything else; accesses to memory are often many orders of magnitude *slower* than other operations (e.g., performing arithmetic). This acts as a barrier, or wall, that prevents computers executing programs quicker that they do. In a rough sense, the reason the memory wall exists is to do with how the electronic components that memory is built from behave. Ideally we would like very large, very fast memories but because of how the electronics work, this is usually impossible: either we can have small, fast memories or large, slow memories. Although we made no reference to this until now, our example computer from Chapter 4 has one of the latter: MEM is large, but slow.

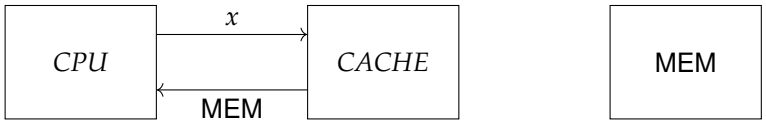
Of course we know computers *do* execute programs quickly, so how has the memory wall being knocked down? The answer is basically a compromise called the **memory hierarchy** [5]. The idea is to have *multiple* memories each representing a different compromise between size and speed, and to use the right one for the right job. This is already evident in our example computer: when we need to store data and access it quickly we use the accumulator $A$, but when we need to store larger quantities of data or for longer periods of time we use MEM. More complicated computer designs have more complicated memory hierarchies; a vital component within such computers is a level of the hierarchy called the **cache** [3], literally a "treasure trove" in which useful data can be hoarded. Cache memories improve performance, but as with everything in life, we seldom get something for nothing. The idea of this Chapter is to explore a *disadvantage* of caches. Specifically, while they improve performance they also introduce a security issue when the programs a computer executes need to prevent leakage of information: we want to demonstrate a side-channel attack based on execution time, putting it within a similar category to the material from Chapter 12.

## 1 Cache memories

A cache is a small, fast memory placed between the computer and MEM: each time the computer tries to access MEM, the access goes through the cache first. Unlike MEM which is "dumb", the cache is "smart" in that some rules govern what it holds. This is important since the cache is smaller than MEM, so it cannot simply hold the same content: instead it just holds *some* of the content in MEM, replicating it with the aim of reducing the total number of times that MEM is accessed.

Imagine the computer performs a memory access using address $x$, e.g., tries to load from MEM. The access goes through the cache, so the cache first checks whether it is currently holding MEM or not. One of two cases will clearly occur:

- If the cache does hold MEM, a **cache-hit** occurs and the value MEM can be returned to the computer quickly and without accessing MEM:




- If the cache does not hold MEM, a **cache-miss** occurs and the value MEM can be returned to the computer only after a subsequent, slower access to MEM:


  Note that because of the size difference, the cache may have to **evict** some of the content it currently holds in order to make space for the new content relating to MEM.

As a short-hand, we use $\mathcal{H}$ and $\mathcal{M}$ to mean a cache-hit and cache-miss where need be. Clearly the hope is that there are lots of cache-hits and few cache-misses. This relies on a property of programs called the **principle of locality**. Imagine you execute a program and pause it at some point in time. Our claim is that whatever the program is doing at that point requires a small **working set** of data relative to the total size of MEM: only a small part of MEM is used at once. Of course this is not *always* true, but for *average* programs there are two good reasons it makes sense; both can be illustrated by an algorithm we saw in Chapter 5. Consider the following,

```
1 algorithm C-String-Length(x) begin
2 |   i ← 0
3 |   while MEM ≠ 0 do
4 |   |   i ← i + 1
5 |   end
6 |   return i
7 end
```

in which two forms of locality can be identified:

**Spacial locality** is where having made one access to memory, there is a good chance the next one will be close to the first. That is, the two accesses use addresses that are close together.

Look at the example: the string is stored in memory, with the algorithm loading characters from MEM. If the $i$-th character is loaded in some iteration of the loop, we know that the next iteration will load the $(i + 1)$-th character which is close by in memory. So accesses to the string are spatially local in this case.

**Temporal locality** is where having made one access to memory, there is a good chance that if you look at another access within a small window of time, it will be the same. That is, the two accesses use the same address.

Look at the example: there is just one assignment in the loop (it adds one to the current value of $i$). One can easily imagine this is achieved using a single instruction; during each iteration of the loop, the computer fetches this instruction from memory in order to then execute it. If the string is $n$ characters long, having fetched the instruction once it will be fetched again $n - 1$ times: only once (i.e., when the loop finishes) will it *not* be fetched again. So accesses to the instruction are temporally local in this case.

The fact that these features are true for average programs basically means that at a given point in time the working set is small: if there is a memory access, there is a good chance it accesses data that is the same as or close to data accessed recently.

There is a more human-centric way to look at the same thing. Psychologists often cite **Miller's Law** [8]: it says that an average human can keep $7 \pm 2$, or between five and nine, items of information (e.g., concepts or facts) in their short-term memory at once. That might seem surprisingly low, so how do we get anything useful done? One factor is the ability for humans to concentrate: they work most effectively when they concentrate on one task. In a way, this is the same thing as locality. What we are saying is that humans can hold a working set of $7 \pm 2$ items of information; their access to information is localised within this working set due to the fact that they concentrate on the related task.

## 1.1 The design of a cache mechanism

Now we have a rough idea about what a cache *should* do, it makes sense to introduce a real design so as to explain what it *does* do. To make sure this explanation is not too overwhelming, we opt to avoid a lot of detail:

1. We consider a simple (perhaps the most simple) cache design.
2. Since both instructions and data are stored in MEM, the cache can hold both as well. However, this makes things more complicated. As a result we will imagine that instructions are fetched directly from MEM (i.e., they bypass the cache) so as to focus on data only.
3. Clearly accesses to MEM can be either loads or stores, but coping with stores is a little more complicated than loads; as a result, we ignore them and focus on loads only.

Numerous resources exist which can flesh out the missing detail if you are interested. The thing to keep in mind is that although we have strayed a little away from reality, our description is close enough to discuss a related security issue later; of course this, rather than the cache itself, is our overall aim so it seems like a good compromise.

So how can we add a cache to our example computer? At the moment, every time the computer needs to load from MEM, we can imagine it following a (very) simple algorithm:

```
1 algorithm LOAD(x) begin
2 |   return MEM
3 end
```

To add a cache, all we really need to think about is how LOAD should be altered so that it applies the appropriate rules using some extra components that we also add.

### 1.1.1 An outline of the cache mechanism

The cache itself is typically organised as a table where each row is called a **cache line**. It is important that we can refer to the different components within a particular cache line, so assume $C$ is the $i$-th cache line:

- $C_{valid}$ indicates whether the $i$-th cache line is valid or not, i.e., whether if there is any content in it or not.
- $C_{tag}$ is the **tag** for the cache line; this acts as identifier for exactly what the cache line contains.
- $C_{data}$ is the actual content, i.e., data, for the $i$-th cache line. This is basically a just sequence whose elements are typically called **sub-words**.

If we want to describe a real cache, it is important we are precise about some of the quantities involved: we assume throughout that there are $l$ cache lines with $w$ sub-words in each. Now we can draw a picture which hopefully makes things a bit clearer:

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 1 | $\langle 4, 1, 7, 5 \rangle$ |
| 1 | **false** | $\perp$ | $\perp$ |
| 2 | **true** | 1 | $\langle 0, 2, 6, 3 \rangle$ |
| 3 | **false** | $\perp$ | $\perp$ |
| 4 | **false** | $\perp$ | $\perp$ |
| 5 | **false** | $\perp$ | $\perp$ |
| 6 | **false** | $\perp$ | $\perp$ |
| 7 | **false** | $\perp$ | $\perp$ |

In this case, $l = 8$ and $w = 4$ so the cache has eight cache lines, each of which can accommodate four sub-words; only the 0-th and 2-nd cache lines have valid content. For example if $C$ refers to the 2-nd cache line, then $C_{valid}$ = **true** and $C_{tag} = 1$; if $D = C_{data}$ then $D_0 = 0$, $D_1 = 2$, $D_2 = 6$ and $D_3 = 3$. Based on this table we can dive in and rewrite the LOAD algorithm:

```
1  algorithm LOAD(x) begin
2  |  x_sub-word ← x mod w
3  |  x_line ← ⌊x/w⌋ mod l
4  |  x_tag ← ⌊(x/w)/l⌋
5  |  x_sub-wordless ← ⌊x/w⌋ · w
6  |  Let C refer to cache line number x_line
7  |  Let D refer to C_data, i.e., the data held in C
8  |  if C_valid = false or C_tag ≠ x_tag then
9  |  |  for i from 0 upto w − 1 do
10 |  |  |  D_i ← MEM
11 |  |  end
12 |  |  C_valid = true
13 |  |  C_tag = x_tag
14 |  end
15 |  return D_x_sub-word
16 end
```

Wow! This is quite a bit more complicated than usual, but we can explain what is going on step-by-step:

- Given the address $x$, lines #2 through #5 perform what is typically called **address translation**: they apply the rules which decide where in the cache the associated data should be stored.

  The idea is fairly simple: MEM is large, but the cache is smaller. Or put another way, there are |MEM| elements in MEM but only a total of $l \cdot w$ sub-words in the cache. We know

  $$l \cdot w < |\text{MEM}|$$

  because that was the point in the first place, i.e., we wanted a smaller, faster version of MEM. So, if we just map the $i$-th element of MEM onto the $i$-th sub-word that would work for the first $l \cdot w$ elements, i.e., elements $0 \ldots l \cdot w - 1$. But what about element $l \cdot w$ which has no corresponding sub-word? To cope, we "wrap around" and map element $l \cdot w$ onto the sub-word 0, element $l \cdot w + 1$ onto sub-word 1 and so on. The assignments to $x_{sub-word}$ and $x_{line}$ decodes $x$ to apply exactly this mapping process.

- The condition construct on line #8 checks if there is valid content in the cache line that $x$ should be in, and whether it is actually the content related to $x$; it does this by checking $C_{valid}$ and then comparing the tag computed from $x$ with the one stored in $C_{tag}$. If both checks pass, we have a cache-hit: the data is there, we just need to access it. If either check fails, we have a cache-miss: before accessing the data we need to fetch it into the cache. Lines #9 through #13 handle this by including a loop construct that loads each sub-word from MEM (we discuss this in more detail below).

  Why would $C_{tag} \neq x_{tag}$? Remember above that we said elements from MEM wrap around: following our reasoning, the 0-th sub-word of cache line 0 could contain element 0 from MEM *or* element $l \cdot w$. The reason we do not get confused between the two is that the tag for element 0 will be different from the tag for element $l \cdot w$. Thus if the comparison fails, we know that although the cache line has valid content in it, that content is not want we are looking for.

- Finally, line #15 returns the data we want to access. Note that by this point we know it must be resident in the cache: either it was there in the first place and there was a cache-hit, or we fetched it due to a cache-miss.

Since we will use the concept later, line #5 is particularly important: the idea is that it calculates a "sub-wordless" version of the address $x$. That is, $x_{sub-wordless}$ is $x$ but without the part that determines the sub-word $x$ should map to. We have $w = 4$ sub-words in each cache line; have a look at what the calculation does if we select $x = 1$ or $x = 34$

$$\begin{array}{lcccccc} x & = & & & 1_{(10)} & = & 000001_{(2)} \\ x_{sub-wordless} & = & \lfloor x/w \rfloor \cdot w & = & 0_{(10)} & = & 000000_{(2)} \\ \\ x & = & & & 34_{(10)} & = & 100010_{(2)} \\ x_{sub-wordless} & = & \lfloor x/w \rfloor \cdot w & = & 32_{(10)} & = & 100000_{(2)} \end{array}$$

Since we have written the result in decimal *and* binary, the process is clearer: to get $x_{sub-wordless}$, we basically set the least-significant (i.e., those on the right-hand side) two bits of $x$ to zero.

This is made use of in line #10 when the cache fetches sub-words from MEM due to a cache-miss. Remember that $x_{sub-wordless}$ is $x$ without the part that determines which sub-word $x$ should map to. By looping through all $i \in \{0, 1, \ldots, w-1\}$, the expression

$$x_{sub-wordless} + i$$

therefore cycles through all the addresses in the cache line $x$ has been mapped into. Consider $x = 34$ again: we know that

$$\begin{array}{lcll} x_{sub-word} & = & x \bmod w & = & 2 \\ x_{line} & = & \lfloor x/w \rfloor \bmod l & = & 0 \\ x_{sub-wordless} & = & \lfloor x/w \rfloor \cdot w & = & 32 \end{array}$$

so address $x = 34$ maps to sub-word 2 of cache line 0. When the loop fetches the corresponding sub-words from MEM, the steps it does are something like

$$\begin{array}{lcll} D_0 & \leftarrow & \mathsf{MEM} & = & \mathsf{MEM} \\ D_1 & \leftarrow & \mathsf{MEM} & = & \mathsf{MEM} \\ D_2 & \leftarrow & \mathsf{MEM} & = & \mathsf{MEM} \\ D_3 & \leftarrow & \mathsf{MEM} & = & \mathsf{MEM} \end{array}$$

which means we end up with the data we wanted in the right place.

### 1.1.2 An example of cache behaviour

The mechanism we described above is (part of) a **direct-mapped cache**. The name basically reflects how the address translation works: an address maps directly to just one one cache line and sub-word. Confused? Imagine we set $l = 8$ and $w = 4$ again as above, and write a program that performs a sequence of nine loads using the addresses

$$1, 34, 35, 36, 37, 1, 38, 39, 40.$$

Such a sequence is often called an **address stream**. Basically we invoke LOAD nine times, i.e., the program execution can be modelled by

$$\begin{array}{lcll} 220001 & \mapsto & A \leftarrow \mathsf{MEM} & \mapsto & \textsc{Load}(1) \\ 220034 & \mapsto & A \leftarrow \mathsf{MEM} & \mapsto & \textsc{Load}(34) \\ 220035 & \mapsto & A \leftarrow \mathsf{MEM} & \mapsto & \textsc{Load}(35) \\ 220036 & \mapsto & A \leftarrow \mathsf{MEM} & \mapsto & \textsc{Load}(36) \\ 220037 & \mapsto & A \leftarrow \mathsf{MEM} & \mapsto & \textsc{Load}(37) \\ 220001 & \mapsto & A \leftarrow \mathsf{MEM} & \mapsto & \textsc{Load}(1) \\ 220038 & \mapsto & A \leftarrow \mathsf{MEM} & \mapsto & \textsc{Load}(38) \\ 220039 & \mapsto & A \leftarrow \mathsf{MEM} & \mapsto & \textsc{Load}(39) \\ 220040 & \mapsto & A \leftarrow \mathsf{MEM} & \mapsto & \textsc{Load}(40) \end{array}$$

Of course there are probably other instructions in there as well, but we can ignore them: from the point of view of the cache, we only care about the loads. One thing worth noting is the address stream already roughly follows the principle of locality: we load from MEM twice (which is temporal locality), and also perform consecutive loads from MEM through MEM and MEM through MEM (which is spacial locality).


In addition to the direct-mapped cache design outlined, other examples also exist; often, one design is preferred over another based on how or where it will be used. Find out about at least one *other* design: explain the motivation for using it, and try to write an algorithm (of a similar style to LOAD) that describes how accesses are satisfied.

The question is, for this program how does the cache behave? In a sense, the content of MEM does not matter: we just want to know when cache-hits and cache-misses occur, not necessarily the values actually loaded. On the other hand, some concrete values might make the example easier to grasp, so imagine that before we execute the program the memory content is as follows:

$$\begin{array}{lcccccccc} i & = & & 0, & 1, & 2, & 3, & \ldots & \\ \mathsf{MEM} & = & \langle & 220001, & 220034, & 220035, & 220036, & \ldots & \rangle \end{array}$$

$$\begin{array}{lcccccccc} i & = & \ldots & 34, & 35, & 36, & 37, & & \\ \mathsf{MEM} & = & \langle \ \ldots & 3, & 1, & 4, & 1, & & \rangle \end{array}$$

$$\begin{array}{lcccccccc} i & = & & 38, & 39, & 40, & 41, & \ldots & \\ \mathsf{MEM} & = & \langle & 5, & 9, & 2, & 6, & \ldots & \rangle \end{array}$$

Now we are ready: from the address stream

$$1, 34, 35, 36, 37, 1, 38, 39, 40$$

we get the cache behaviour

$$\mathcal{M}, \mathcal{M}, \mathcal{H}, \mathcal{M}, \mathcal{H}, \mathcal{M}, \mathcal{H}, \mathcal{H}, \mathcal{M},$$

i.e., a cache-miss followed by another cache-miss, then a cache-hit and so on. Of course this is not much of an explanation; tracing through each step in the algorithm invocations is a little tedious, but here is at least a little more detailed account of what the cache does:

**Step** #1**:** Reset the cache to an initial, empty state.

**Step** #2**:** Address 1 decoded into sub-word 1, line 0 and tag 0; this implies a cache-miss (the content of line #0 is not valid). Line #0 is filled by fetching MEM[0] through MEM[3]. Finally, the value 220034 is loaded from line #0, sub-word #1 which acts as an alias for MEM[1].

**Step** #3**:** Address 34 decoded into sub-word 2, line 0 and tag 1; this implies a cache-miss (it did not match the tag in line #0). Resident content of line #0 is evicted, and refilled by fetching MEM[32] through MEM[35]. Finally, the value 3 is loaded from line #0, sub-word #2 which acts as an alias for MEM[34].

**Step** #4**:** Address 35 decoded into sub-word 3, line 0 and tag 1; this implies a cache-hit (it matched the tag in line #0). Finally, the value 1 is loaded from line #0, sub-word #3 which acts as an alias for MEM[35].

**Step** #5**:** Address 36 decoded into sub-word 0, line 1 and tag 1; this implies a cache-miss (the content of line #1 is not valid). Line #1 is filled by fetching MEM[36] through MEM[39]. Finally, the value 4 is loaded from line #1, sub-word #0 which acts as an alias for MEM[36].

**Step** #6**:** Address 37 decoded into sub-word 1, line 1 and tag 1; this implies a cache-hit (it matched the tag in line #1). Finally, the value 1 is loaded from line #1, sub-word #1 which acts as an alias for MEM[37].

**Step** #7**:** Address 1 decoded into sub-word 1, line 0 and tag 0; this implies a cache-miss (it did not match the tag in line #0). Resident content of line #0 is evicted, and refilled by fetching MEM[0] through MEM[3]. Finally, the value 220034 is loaded from line #0, sub-word #1 which acts as an alias for MEM[1].

**Step** #8**:** Address 38 decoded into sub-word 2, line 1 and tag 1; this implies a cache-hit (it matched the tag in line #1). Finally, the value 5 is loaded from line #1, sub-word #2 which acts as an alias for MEM[38].

**Step** #9**:** Address 39 decoded into sub-word 3, line 1 and tag 1; this implies a cache-hit (it matched the tag in line #1). Finally, the value 9 is loaded from line #1, sub-word #3 which acts as an alias for MEM[39].

**Step** #10**:** Address 40 decoded into sub-word 0, line 2 and tag 1; this implies a cache-miss (the content of line #2 is not valid). Line #2 is filled by fetching MEM[40] through MEM[43]. Finally, the value 2 is loaded from line #2, sub-word #0 which acts as an alias for MEM[40].

The same thing can be visualised as a series of diagrams, such as those in Figure 1 that show the state of the cache after each of the loads has been performed.

Interestingly, although we said that the address stream looked good in terms of locality the result is not that good: we get the same number of cache-hits as cache-misses in this case. In part this is because the access stream creates a lot of **interference**. This is where accesses evict each other from the cache, by virtue of the rules it applies, even though keeping the corresponding content there would be of more benefit.

## 1.2 Reasoning about cache effectiveness

Imagine we write a program and execute it on our newly upgraded computer; say there are $m$ memory accesses in total, of which $m_{\mathcal{H}}$ result in cache-hits and $m_{\mathcal{M}}$ result in cache-misses. This means

$$m_{\mathcal{H}} + m_{\mathcal{M}} = m.$$

Using these quantities we can measure the proportion of accesses which were cache-hits or cache-misses; we call these the **hit ratio** and **miss ratio**, and write them as

$$\begin{aligned} R_{\mathcal{H}} &= \frac{m_{\mathcal{H}}}{m} \\ R_{\mathcal{M}} &= \frac{m_{\mathcal{M}}}{m} \end{aligned}$$

where clearly now

$$R_{\mathcal{H}} + R_{\mathcal{M}} = 1.$$

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 1 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 2 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(a)** *Step* #1*: Initial, empty state.*

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 0 | ⟨MEM[0], MEM[1], MEM[2], MEM[3]⟩ |
| 1 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 2 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(b)** *Step* #2*: State after load of* 220034 *from address* 1.

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 1 | ⟨MEM[32], MEM[33], MEM[34], MEM[35]⟩ |
| 1 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 2 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(c)** *Step* #3*: State after load of* 3 *from address* 34.

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 1 | ⟨MEM[32], MEM[33], MEM[34], MEM[35]⟩ |
| 1 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 2 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(d)** *Step* #4*: State after load of* 1 *from address* 35.

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 1 | ⟨MEM[32], MEM[33], MEM[34], MEM[35]⟩ |
| 1 | **true** | 1 | ⟨MEM[36], MEM[37], MEM[38], MEM[39]⟩ |
| 2 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(e)** *Step* #5*: State after load of* 4 *from address* 36.

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 1 | ⟨MEM[32], MEM[33], MEM[34], MEM[35]⟩ |
| 1 | **true** | 1 | ⟨MEM[36], MEM[37], MEM[38], MEM[39]⟩ |
| 2 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(f)** *Step* #6*: State after load of* 1 *from address* 37.

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 0 | ⟨MEM[0], MEM[1], MEM[2], MEM[3]⟩ |
| 1 | **true** | 1 | ⟨MEM[36], MEM[37], MEM[38], MEM[39]⟩ |
| 2 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(g)** *Step* #7*: State after load of* 220034 *from address* 1.

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 0 | ⟨MEM[0], MEM[1], MEM[2], MEM[3]⟩ |
| 1 | **true** | 1 | ⟨MEM[36], MEM[37], MEM[38], MEM[39]⟩ |
| 2 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(h)** *Step* #8*: State after load of* 5 *from address* 38.

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 0 | ⟨MEM[0], MEM[1], MEM[2], MEM[3]⟩ |
| 1 | **true** | 1 | ⟨MEM[36], MEM[37], MEM[38], MEM[39]⟩ |
| 2 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(i)** *Step* #9*: State after load of* 9 *from address* 39.

| CACHE | | | |
|---|---|---|---|
| *Line* | *Valid* | *Tag* | *Data* |
| 0 | **true** | 0 | ⟨MEM[0], MEM[1], MEM[2], MEM[3]⟩ |
| 1 | **true** | 1 | ⟨MEM[36], MEM[37], MEM[38], MEM[39]⟩ |
| 2 | **true** | 1 | ⟨MEM[40], MEM[41], MEM[42], MEM[43]⟩ |
| 3 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 4 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 5 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 6 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |
| 7 | **false** | ⊥ | ⟨⊥, ⊥, ⊥, ⊥⟩ |

**(j)** *Step* #10*: State after load of* 2 *from address* 40.

**Figure 1:** *The behaviour of a small direct-mapped cache when a sequence of loads from addresses* 1, 34, 35, 36, 37, 1, 38, 39 *and* 40 *is performed.*

Another way to think of these quantities is as probabilities: if $m$ is large enough, then our reasoning is that $R_{\mathcal{H}}$ and $R_{\mathcal{M}}$ are good general estimates as well as specific results for our program. Remember that our program is average in some sense, so basically we can say that on average an access is a cache-hit with probability $R_{\mathcal{H}}$ or conversely a cache-miss with probability $R_{\mathcal{M}}$.

From the hit and miss ratios, we can start to reason about **mean access time**, i.e., the average time it takes to perform a memory access. When there was no cache, this was simple: all we needed to do was load from or store into MEM. We can think the time taken to do this as being described by a constant $T_{\mathsf{MEM}}$. Now we have a cache, the time taken depends on whether the access was a cache-hit or a cache-miss. But this is not too hard to accommodate: remember we want the mean (or average) access time so our probabilities from above become useful.

Imagine the time taken to access the cache is $T_{CACHE}$. Obviously an access can either cause a cache-hit or a cache-miss, so the mean access time (the average time taken for such an access) is given in two parts:

- For a cache-hit, the element of MEM we are interested in is already resident in the cache; we just need to load from or store into it. Since an access is a cache-hit with probability $R_{\mathcal{H}}$, the time taken on average will be
$$R_{\mathcal{H}} \cdot T_{CACHE}.$$
- For a cache-miss, the element of MEM we are interested in is not resident in the cache; first we just need to fetch it from MEM into the cache and then load from or store into it once it becomes resident. Since the access is a cache-miss with probability $R_{\mathcal{M}}$, the time taken on average will be
$$R_{\mathcal{M}} \cdot (T_{\mathsf{MEM}} + T_{CACHE}).$$

The mean access time is therefore the sum of these, i.e.,

$$T = R_{\mathcal{H}} \cdot T_{CACHE} + R_{\mathcal{M}} \cdot (T_{\mathsf{MEM}} + T_{CACHE}),$$

or in words: the probability of the access being a cache-hit multiplied by the time taken to deal with a cache-hit, plus the probability of the access being a cache-miss multiplied by the time taken to deal with a cache-miss.

We can do some analysis and see if this makes sense. If the hit ratio tends to 1 (meaning the miss ratio tends to 0), the first term above dominates and the mean access time tends to $T_{CACHE}$. This is good: if there are lots of cache-hits, $T$ will be small so the time taken to perform the associated accesses will be small. However, if the hit ratio tends to 0 (meaning the miss ratio tends to 1), the second term above dominates and the mean access time now tends to $T_{\mathsf{MEM}} + T_{CACHE}$. This is bad: if there are lots of cache-misses, $T$ will be large so the time taken to perform the associated accesses will be large. In fact this is *very* bad because if the mean access time is $T_{\mathsf{MEM}} + T_{CACHE}$ then clearly this is larger than $T_{\mathsf{MEM}}$ and so we are actually slower than if the cache was removed entirely!

# 2 On straight-line execution

## 2.1 Applying specialisation and loop unrolling

Imagine we write an algorithm with a loop in it; the idea of the algorithm is to set the first $n$ elements in a sequence called $A$ to zero:

```
1 algorithm Zero(A, n) begin
2 |   for i from 0 upto n − 1 do
3 |   |   A_i ← 0
4 |   end
5 |   return
6 end
```

The steps performed when Zero is invoked are easy to work out. Writing them out in a more verbose way than before, we would do something like the following:

**Step** #1 Assign $i \leftarrow 0$.

**Step** #2 Since $i \leq n - 1$, perform the next loop iteration.

**Step** #3 Assign $A_i \leftarrow 0$, i.e., $A_0 \leftarrow 0$.

**Step** #4 Assign $i \leftarrow i + 1$, i.e., $i \leftarrow 1$.

**Step** #5 Since $i \leq n - 1$, perform the next loop iteration.

**Step** #6 Assign $A_i \leftarrow 0$, i.e., $A_1 \leftarrow 0$.

**Step** #7 Assign $i \leftarrow i + 1$, i.e., $i \leftarrow 2$.

**Step** #8 Since $i \leq n - 1$, perform the next loop iteration.

**Step** #9 Assign $A_i \leftarrow 0$, i.e., $A_2 \leftarrow 0$.

**Step** #10 Assign $i \leftarrow i + 1$, i.e., $i \leftarrow 3$.

**Step** #11 Since $i \leq n - 1$, perform the next loop iteration.

**Step** #12 Assign $A_i \leftarrow 0$, i.e., $A_3 \leftarrow 0$.

**Step** #13 Assign $i \leftarrow i + 1$, i.e., $i \leftarrow 4$.

**Step** #14 ...

Eventually, $i = n$ and the loop stops. It might seem odd that we do so many steps yet accomplish so little: where have the extra steps come from? The answer is obvious in the sense that the extra steps relate to control of the loop. Although the block we iterate over (i.e., line #3 of the algorithm) needs one step per-iteration, we need an extra two steps to test and increment $i$. Since we do $n$ iterations, the number of steps is therefore roughly $3n$ rather than just $n$.

Clearly this is unattractive: the more steps the algorithm takes, the longer it takes to give us a result. This is especially annoying if we know the number of iterations which should be performed before we start. In this situation the loop just acts like a short description for a potentially long sequence of steps. We can resolve this problem by in two steps. First imagine $n = 4$, though any value will do. Using this choice, we can **specialise** the original algorithm:

```
1 algorithm ZERO(A) begin
2 |   for i from 0 upto 3 do
3 |   |   A_i ← 0
4 |   end
5 |   return
6 end
```

Notice that now the algorithm is a special-purpose version, the loop now *always* does $n = 4$ iterations. We cannot ask it to do more, or less, by changing $n$ like we could with the original one: $n$ is now fixed. On one hand this does not give us any advantage because this loop still needs about $3n = 12$ steps, just like the original one. On the other hand, now it is clear how many iterations the loop should perform, we can **unroll** it. The idea is to take the block representing the loop body and copy it $n$ times, each time replacing $i$ with the right value. The end result is:

```
1 algorithm ZERO(A) begin
2 |   A_0 ← 0
3 |   A_1 ← 0
4 |   A_2 ← 0
5 |   A_3 ← 0
6 |   return
7 end
```

which now only takes four steps to do the same thing!

### 2.1.1 Estimating execution time

Looking at the discussion above, the aim of unrolling a loop seems to be an improvement in performance, i.e., a reduction in steps. This is true, but it also gives a second advantage: the fact that the unrolling process yields a **straight-line** version (i.e., just a sequence of assignments) means we can reason about it more easily.

Imagine we have a straight-line version of some algorithm, and implement it as a program with $m$ instructions in it. We can estimate the execution time of the program using

$$T = m \cdot T_{EXE}$$

where $T_{EXE}$ is a constant used to represent how long a single instruction takes to execute. If we *really* wanted to be accurate, we might have a different value of $T_{EXE}$ for each type of instruction. This would allow us to capture the idea that some instructions might take longer to execute than others for example.

This is not the end of the story however: remember that some instructions will access memory and when they do so, the cache plays a part. Put more specifically, some memory accesses will provoke cache-hits and others will provoke cache-misses. Imagine there are $n \leq m$ instructions that access memory. Based on what we already know about mean access time for such instructions, we can rewrite out estimate for the time the program takes to execute as

$$T = m \cdot T_{EXE} + n \cdot (R_{\mathcal{H}} \cdot T_{CACHE} + R_{\mathcal{M}} \cdot (T_{MEM} + T_{CACHE})).$$

Now, the total is now the time taken to execute all the instructions plus the overhead of accessing memory: the latter term is simply the mean access time for a single access (which we already worked out) multiplied by $n$, the number of accesses.

A concrete example will probably make this much clearer. Think back to the straight-line version of the Zero algorithm which sets four elements of a sequence to zero, and imagine we implement the corresponding program. Given we are going to execute the program on a real computer, we need to given concrete values to the constants in our estimations. Imagine that

$$\begin{array}{lcl} T_{EXE} & = & 1 \\ T_{CACHE} & = & 5 \\ T_{MEM} & = & 10 \end{array}$$

so that accessing the cache takes exactly half the time of memory, and both are quite a bit larger than the time taken to execute instructions. There are five instructions in the program; the first four set elements of the sequence to zero. So using our initial estimate we get

$$\begin{array}{rcl} T & = & 5 \cdot T_{EXE} \\ & = & 5 \cdot 1 \\ & = & 5 \end{array}$$

But the first four instructions all access memory, so to be more accurate we should use the better estimation method. To do that we need to fix the hit and miss ratio of out cache; imagine $R_{\mathcal{H}} = 0.5$ and therefore $R_{\mathcal{M}} = 0.5$, so a healthy 50% of accesses will cause cache-hits on average. This means

$$\begin{array}{rcll} T & = & 5 \cdot T_{EXE} & + \quad 4 \cdot (R_{\mathcal{H}} \cdot T_{CACHE} + R_{\mathcal{M}} \cdot (T_{MEM} + T_{CACHE})) \\ & = & 5 \cdot 1 & + \quad 4 \cdot (0.5 \cdot 5 + 0.5 \cdot (10 + 5)) \\ & = & 45 & \end{array}$$

What happens if the cache is designed to be more effective: now $R_{\mathcal{H}} = 0.8$ and therefore $R_{\mathcal{H}} = 0.2$, so now 80% of accesses will cause cache-hits on average. Clearly this alters our estimate

$$\begin{array}{rcll} T & = & 5 \cdot T_{EXE} & + \quad 4 \cdot (R_{\mathcal{H}} \cdot T_{CACHE} + R_{\mathcal{M}} \cdot (T_{MEM} + T_{CACHE})) \\ & = & 5 \cdot 1 & + \quad 4 \cdot (0.8 \cdot 5 + 0.2 \cdot (10 + 5)) \\ & = & 33 & \end{array}$$

in that now the program executes more quickly. In fact, the more cache-hits we get, the faster the program will execute. This is good news because this was the whole point of putting it there in the first place! The key thing to notice is that the memory accesses are what cause variation in our straight-line program: the number of instructions in the program is fixed, as are the constants such as $T_{EXE}$, so if the execution time varies we *know* that cache behaviour is the cause.

```
1  algorithm ENC(K, P, n, r) begin
2      for i from 0 upto r − 1 do
3          for j from 0 upto n − 1 do
4              P_j ← ET(P_j ⊕ K_j)
5          end
6          t ← P_0
7          for j from 1 upto n − 1 do
8              P_{j−1} ← P_j
9          end
10         P_{n−1} ← t
11     end
12     return
13 end
```

**(a)** *An algorithm describing* ENC.

```
1  algorithm DEC(K, C, n, r) begin
2      for i from 0 upto r − 1 do
3          t ← C_{n−1}
4          for j from n − 1 downto 1 do
5              C_j ← C_{j−1}
6          end
7          C_0 ← t
8          for j from 0 upto n − 1 do
9              C_j ← DT(C_j) ⊕ K_j
10         end
11     end
12     return
13 end
```

**(b)** *An algorithm describing* DEC.

**Figure 2:** *An example cipher, somewhat similar to AES, which acts as the target of an example cache-based side-channel attack.*

```
1  algorithm ENC(K, P) begin
2      P_0 ← ET-TAB_{P_0⊕K_0}
3      P_1 ← ET-TAB_{P_1⊕K_1}
4      t ← P_0
5      P_0 ← P_1
6      P_1 ← t
7      P_0 ← ET-TAB_{P_0⊕K_0}
8      P_1 ← ET-TAB_{P_1⊕K_1}
9      t ← P_0
10     P_0 ← P_1
11     P_1 ← t
12     return P
13 end
```

**(a)** *An algorithm describing* ENC.

```
1  algorithm DEC(K, C) begin
2      t ← C_1
3      C_1 ← C_0
4      C_0 ← t
5      C_0 ← DT-TAB_{C_0} ⊕ K_0
6      C_1 ← DT-TAB_{C_1} ⊕ K_1
7      t ← C_1
8      C_1 ← C_0
9      C_0 ← t
10     C_0 ← DT-TAB_{C_0} ⊕ K_0
11     C_1 ← DT-TAB_{C_1} ⊕ K_1
12     return C
13 end
```

**(b)** *An algorithm describing* DEC.

**Figure 3:** *The original example cipher specialised and unrolled for $n = 2$ and $r = 2$ to yield simpler descriptions, and now using the* ET-TAB *and* DT-TAB *tables rather than the* ET *and* DT *functions.*

$$
\begin{aligned}
\text{ET-Tab} = \langle\ & 63_{(16)}, 7C_{(16)}, 77_{(16)}, 7B_{(16)}, F2_{(16)}, 6B_{(16)}, 6F_{(16)}, C5_{(16)},\\
& 30_{(16)}, 01_{(16)}, 67_{(16)}, 2B_{(16)}, FE_{(16)}, D7_{(16)}, AB_{(16)}, 76_{(16)},\\
& CA_{(16)}, 82_{(16)}, C9_{(16)}, 7D_{(16)}, FA_{(16)}, 59_{(16)}, 47_{(16)}, F0_{(16)},\\
& AD_{(16)}, D4_{(16)}, A2_{(16)}, AF_{(16)}, 9C_{(16)}, A4_{(16)}, 72_{(16)}, C0_{(16)},\\
& B7_{(16)}, FD_{(16)}, 93_{(16)}, 26_{(16)}, 36_{(16)}, 3F_{(16)}, F7_{(16)}, CC_{(16)},\\
& 34_{(16)}, A5_{(16)}, E5_{(16)}, F1_{(16)}, 71_{(16)}, D8_{(16)}, 31_{(16)}, 15_{(16)},\\
& 04_{(16)}, C7_{(16)}, 23_{(16)}, C3_{(16)}, 18_{(16)}, 96_{(16)}, 05_{(16)}, 9A_{(16)},\\
& 07_{(16)}, 12_{(16)}, 80_{(16)}, E2_{(16)}, EB_{(16)}, 27_{(16)}, B2_{(16)}, 75_{(16)},\\
& 09_{(16)}, 83_{(16)}, 2C_{(16)}, 1A_{(16)}, 1B_{(16)}, 6E_{(16)}, 5A_{(16)}, A0_{(16)},\\
& 52_{(16)}, 3B_{(16)}, D6_{(16)}, B3_{(16)}, 29_{(16)}, E3_{(16)}, 2F_{(16)}, 84_{(16)},\\
& 53_{(16)}, D1_{(16)}, 00_{(16)}, ED_{(16)}, 20_{(16)}, FC_{(16)}, B1_{(16)}, 5B_{(16)},\\
& 6A_{(16)}, CB_{(16)}, BE_{(16)}, 39_{(16)}, 4A_{(16)}, 4C_{(16)}, 58_{(16)}, CF_{(16)},\\
& D0_{(16)}, EF_{(16)}, AA_{(16)}, FB_{(16)}, 43_{(16)}, 4D_{(16)}, 33_{(16)}, 85_{(16)},\\
& 45_{(16)}, F9_{(16)}, 02_{(16)}, 7F_{(16)}, 50_{(16)}, 3C_{(16)}, 9F_{(16)}, A8_{(16)},\\
& 51_{(16)}, A3_{(16)}, 40_{(16)}, 8F_{(16)}, 92_{(16)}, 9D_{(16)}, 38_{(16)}, F5_{(16)},\\
& BC_{(16)}, B6_{(16)}, DA_{(16)}, 21_{(16)}, 10_{(16)}, FF_{(16)}, F3_{(16)}, D2_{(16)},\\
& CD_{(16)}, 0C_{(16)}, 13_{(16)}, EC_{(16)}, 5F_{(16)}, 97_{(16)}, 44_{(16)}, 17_{(16)},\\
& C4_{(16)}, A7_{(16)}, 7E_{(16)}, 3D_{(16)}, 64_{(16)}, 5D_{(16)}, 19_{(16)}, 73_{(16)},\\
& 60_{(16)}, 81_{(16)}, 4F_{(16)}, DC_{(16)}, 22_{(16)}, 2A_{(16)}, 90_{(16)}, 88_{(16)},\\
& 46_{(16)}, EE_{(16)}, B8_{(16)}, 14_{(16)}, DE_{(16)}, 5E_{(16)}, 0B_{(16)}, DB_{(16)},\\
& E0_{(16)}, 32_{(16)}, 3A_{(16)}, 0A_{(16)}, 49_{(16)}, 06_{(16)}, 24_{(16)}, 5C_{(16)},\\
& C2_{(16)}, D3_{(16)}, AC_{(16)}, 62_{(16)}, 91_{(16)}, 95_{(16)}, E4_{(16)}, 79_{(16)},\\
& E7_{(16)}, C8_{(16)}, 37_{(16)}, 6D_{(16)}, 8D_{(16)}, D5_{(16)}, 4E_{(16)}, A9_{(16)},\\
& 6C_{(16)}, 56_{(16)}, F4_{(16)}, EA_{(16)}, 65_{(16)}, 7A_{(16)}, AE_{(16)}, 08_{(16)},\\
& BA_{(16)}, 78_{(16)}, 25_{(16)}, 2E_{(16)}, 1C_{(16)}, A6_{(16)}, B4_{(16)}, C6_{(16)},\\
& E8_{(16)}, DD_{(16)}, 74_{(16)}, 1F_{(16)}, 4B_{(16)}, BD_{(16)}, 8B_{(16)}, 8A_{(16)},\\
& 70_{(16)}, 3E_{(16)}, B5_{(16)}, 66_{(16)}, 48_{(16)}, 03_{(16)}, F6_{(16)}, 0E_{(16)},\\
& 61_{(16)}, 35_{(16)}, 57_{(16)}, B9_{(16)}, 86_{(16)}, C1_{(16)}, 1D_{(16)}, 9E_{(16)},\\
& E1_{(16)}, F8_{(16)}, 98_{(16)}, 11_{(16)}, 69_{(16)}, D9_{(16)}, 8E_{(16)}, 94_{(16)},\\
& 9B_{(16)}, 1E_{(16)}, 87_{(16)}, E9_{(16)}, CE_{(16)}, 55_{(16)}, 28_{(16)}, DF_{(16)},\\
& 8C_{(16)}, A1_{(16)}, 89_{(16)}, 0D_{(16)}, BF_{(16)}, E6_{(16)}, 42_{(16)}, 68_{(16)},\\
& 41_{(16)}, 99_{(16)}, 2D_{(16)}, 0F_{(16)}, B0_{(16)}, 54_{(16)}, BB_{(16)}, 16_{(16)}\ \rangle
\end{aligned}
$$

**Figure 4:** ET-Tab, *a tabular description of* ET.

$$
\begin{aligned}
\text{DT-TAB} = \langle\ & 52_{(16)}, 09_{(16)}, 6A_{(16)}, D5_{(16)}, 30_{(16)}, 36_{(16)}, A5_{(16)}, 38_{(16)}, \\
& BF_{(16)}, 40_{(16)}, A3_{(16)}, 9E_{(16)}, 81_{(16)}, F3_{(16)}, D7_{(16)}, FB_{(16)}, \\
& 7C_{(16)}, E3_{(16)}, 39_{(16)}, 82_{(16)}, 9B_{(16)}, 2F_{(16)}, FF_{(16)}, 87_{(16)}, \\
& 34_{(16)}, 8E_{(16)}, 43_{(16)}, 44_{(16)}, C4_{(16)}, DE_{(16)}, E9_{(16)}, CB_{(16)}, \\
& 54_{(16)}, 7B_{(16)}, 94_{(16)}, 32_{(16)}, A6_{(16)}, C2_{(16)}, 23_{(16)}, 3D_{(16)}, \\
& EE_{(16)}, 4C_{(16)}, 95_{(16)}, 0B_{(16)}, 42_{(16)}, FA_{(16)}, C3_{(16)}, 4E_{(16)}, \\
& 08_{(16)}, 2E_{(16)}, A1_{(16)}, 66_{(16)}, 28_{(16)}, D9_{(16)}, 24_{(16)}, B2_{(16)}, \\
& 76_{(16)}, 5B_{(16)}, A2_{(16)}, 49_{(16)}, 6D_{(16)}, 8B_{(16)}, D1_{(16)}, 25_{(16)}, \\
& 72_{(16)}, F8_{(16)}, F6_{(16)}, 64_{(16)}, 86_{(16)}, 68_{(16)}, 98_{(16)}, 16_{(16)}, \\
& D4_{(16)}, A4_{(16)}, 5C_{(16)}, CC_{(16)}, 5D_{(16)}, 65_{(16)}, B6_{(16)}, 92_{(16)}, \\
& 6C_{(16)}, 70_{(16)}, 48_{(16)}, 50_{(16)}, FD_{(16)}, ED_{(16)}, B9_{(16)}, DA_{(16)}, \\
& 5E_{(16)}, 15_{(16)}, 46_{(16)}, 57_{(16)}, A7_{(16)}, 8D_{(16)}, 9D_{(16)}, 84_{(16)}, \\
& 90_{(16)}, D8_{(16)}, AB_{(16)}, 00_{(16)}, 8C_{(16)}, BC_{(16)}, D3_{(16)}, 0A_{(16)}, \\
& F7_{(16)}, E4_{(16)}, 58_{(16)}, 05_{(16)}, B8_{(16)}, B3_{(16)}, 45_{(16)}, 06_{(16)}, \\
& D0_{(16)}, 2C_{(16)}, 1E_{(16)}, 8F_{(16)}, CA_{(16)}, 3F_{(16)}, 0F_{(16)}, 02_{(16)}, \\
& C1_{(16)}, AF_{(16)}, BD_{(16)}, 03_{(16)}, 01_{(16)}, 13_{(16)}, 8A_{(16)}, 6B_{(16)}, \\
& 3A_{(16)}, 91_{(16)}, 11_{(16)}, 41_{(16)}, 4F_{(16)}, 67_{(16)}, DC_{(16)}, EA_{(16)}, \\
& 97_{(16)}, F2_{(16)}, CF_{(16)}, CE_{(16)}, F0_{(16)}, B4_{(16)}, E6_{(16)}, 73_{(16)}, \\
& 96_{(16)}, AC_{(16)}, 74_{(16)}, 22_{(16)}, E7_{(16)}, AD_{(16)}, 35_{(16)}, 85_{(16)}, \\
& E2_{(16)}, F9_{(16)}, 37_{(16)}, E8_{(16)}, 1C_{(16)}, 75_{(16)}, DF_{(16)}, 6E_{(16)}, \\
& 47_{(16)}, F1_{(16)}, 1A_{(16)}, 71_{(16)}, 1D_{(16)}, 29_{(16)}, C5_{(16)}, 89_{(16)}, \\
& 6F_{(16)}, B7_{(16)}, 62_{(16)}, 0E_{(16)}, AA_{(16)}, 18_{(16)}, BE_{(16)}, 1B_{(16)}, \\
& FC_{(16)}, 56_{(16)}, 3E_{(16)}, 4B_{(16)}, C6_{(16)}, D2_{(16)}, 79_{(16)}, 20_{(16)}, \\
& 9A_{(16)}, DB_{(16)}, C0_{(16)}, FE_{(16)}, 78_{(16)}, CD_{(16)}, 5A_{(16)}, F4_{(16)}, \\
& 1F_{(16)}, DD_{(16)}, A8_{(16)}, 33_{(16)}, 88_{(16)}, 07_{(16)}, C7_{(16)}, 31_{(16)}, \\
& B1_{(16)}, 12_{(16)}, 10_{(16)}, 59_{(16)}, 27_{(16)}, 80_{(16)}, EC_{(16)}, 5F_{(16)}, \\
& 60_{(16)}, 51_{(16)}, 7F_{(16)}, A9_{(16)}, 19_{(16)}, B5_{(16)}, 4A_{(16)}, 0D_{(16)}, \\
& 2D_{(16)}, E5_{(16)}, 7A_{(16)}, 9F_{(16)}, 93_{(16)}, C9_{(16)}, 9C_{(16)}, EF_{(16)}, \\
& A0_{(16)}, E0_{(16)}, 3B_{(16)}, 4D_{(16)}, AE_{(16)}, 2A_{(16)}, F5_{(16)}, B0_{(16)}, \\
& C8_{(16)}, EB_{(16)}, BB_{(16)}, 3C_{(16)}, 83_{(16)}, 53_{(16)}, 99_{(16)}, 61_{(16)}, \\
& 17_{(16)}, 2B_{(16)}, 04_{(16)}, 7E_{(16)}, BA_{(16)}, 77_{(16)}, D6_{(16)}, 26_{(16)}, \\
& E1_{(16)}, 69_{(16)}, 14_{(16)}, 63_{(16)}, 55_{(16)}, 21_{(16)}, 0C_{(16)}, 7D_{(16)}\ \rangle
\end{aligned}
$$

**Figure 5:** DT-TAB, *a tabular description of* DT.

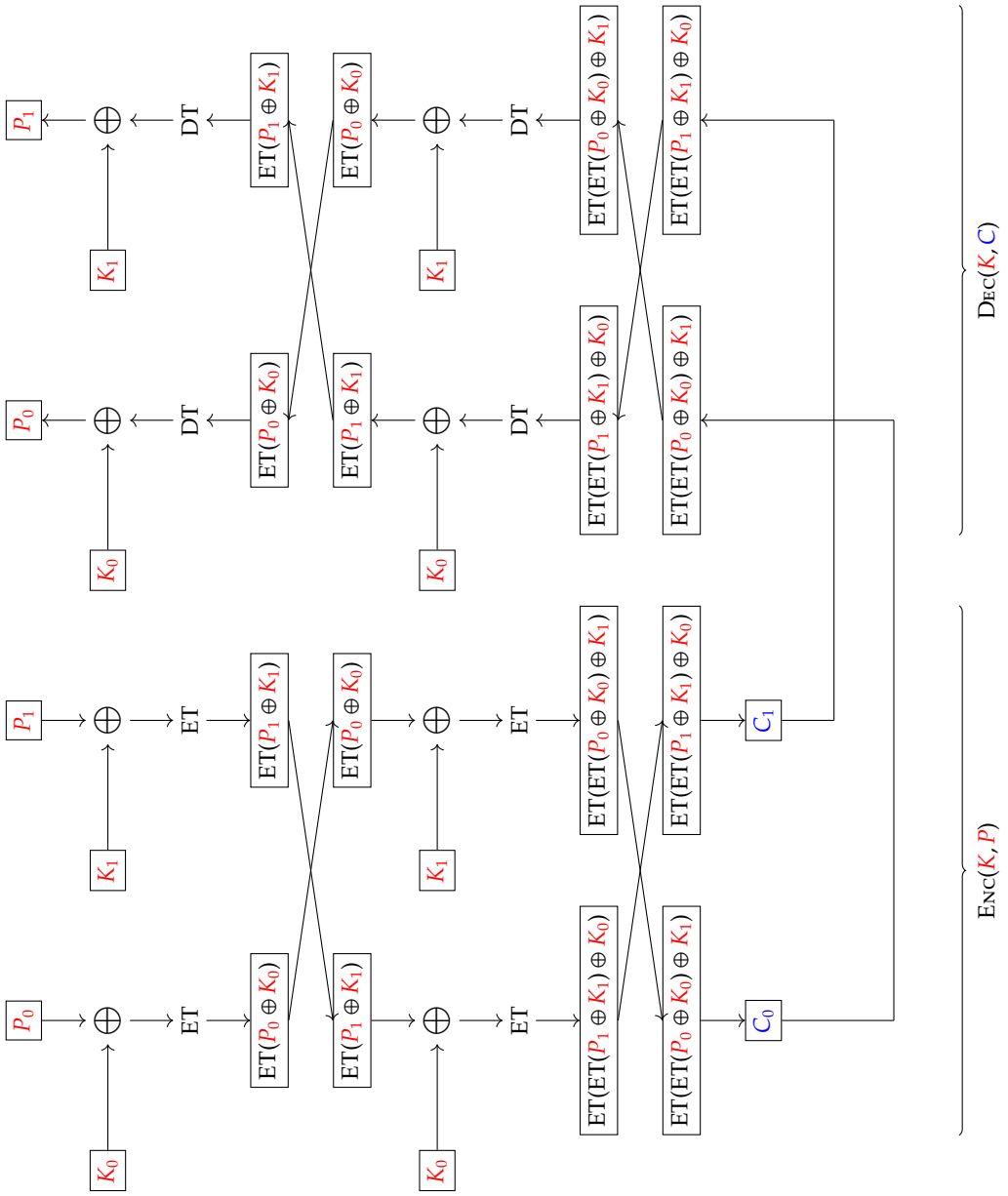



**Figure 6:** *A diagrammatic description of* Enc *and* Dec *invoked using* $n = 2$ *and* $r = 2$*; the left-hand portion represents the encryption of the plaintext message* $P$ *using the key* $K$*, while the right-hand portion represents decryption of the resulting ciphertext* $C$*.*

# 3 An example block cipher

Discussing an attack without discussing the target of said attack is a bit pointless: we need some cryptography to apply the attack to! Now would be a good time to go back and recap on Chapter 7 and Chapter 10. We need two functions which constitute a **block cipher** [2]:

- ENC encrypts a plaintext message $P$ using a key $K$ to produce a ciphertext $C$, and
- DEC decrypts a ciphertext message $C$ using a key $K$ to produce a plaintext $P$.

To make the cipher work, we need to ensure

$$\text{DEC}(K, \text{ENC}(K, P)) = P$$

i.e., DEC is the **inverse function** [4] of ENC for a given key, but clearly only someone who knows the key $K$ can use either of them.

To specify an example cipher, we need only specify how the ENC and DEC functions work; Figure 2 includes two algorithms which do just that. We will focus on ENC: it encrypts a plaintext message $P$ using a key $K$, both of which are just $n$-element sequences of numbers. It certainly has flaws, but one major advantage of this example is that the structure resembles a *real* block cipher. In particular, it looks similar to the **Advanced Encryption Standard (AES)** [1] which is now the de facto standard choice made in most real applications. AES is based on a **Substitution-Permutation network** [7] or "SP-network": it performs substitutions (i.e., replaces elements with other elements) and permutations (i.e., rearranges elements), and also mix in key material, within each of several repeated **rounds**. Looking at Figure 2a for example, we can identify similar features:

- Lines #3 to #5 mix each element from $P$ with one from $K$, and then replaces each element in $P$ with another element.
- Lines #6 to #10 rearrange the elements in $P$.
- Lines #3 to #10 represent a single round which is applied $r$ times by the outer-loop.

Of course there are massive differences as well, but hopefully this adds at least some authenticity: we have an example which is not too far away from reality.

AES very roughly relates to the use of $n = 16$ and $r = 10$: for a 16-element $P$, it applies 10 rounds of processing using a $K$ of the same size. We will make life significantly easier by using $n = 2$ and $r = 2$ for our example. One advantage of this is that we can draw Figure 6 to show how the example cipher works visually. Starting at the top left-hand corner and reading downwards, the diagram applies the steps of ENC; reading from the bottom right-hand corner upwards applies the steps of DEC. Notice how each step of decryption "undoes" the corresponding step of encryption; to explain why this works, we need to explain what the mysterious ET and DT functions are.

## 3.1 Applying the substitution step using an S-box

The substitution steps are particularly important. The idea is that during encryption (resp. decryption) we replace the $j$-th element of $P$ (resp. $C$) by applying a function ET (resp. DT). The two functions are each others inverse so

$$\text{DT}(\text{ET}(x)) = x$$

and

$$\text{ET}(\text{DT}(x)) = x$$

for any $x$. These functions are given a special name; they are called an **S-box** [6] or "substitution box". For our purposes it does not matter what what result ET(255) produces for example, the only thing that matters is the functions act as each others inverse. So to continue with our aim of staying as close to AES as we can, we borrow the AES S-box. There are quite complicated ways to describe the AES S-box, but a much easier way is to simply write out all the entries. That is, we write out two sequences (or rather tables) where

$$\begin{aligned} \text{ET}(x) &= \text{ET-TAB}_x \\ \text{DT}(x) &= \text{DT-TAB}_x \end{aligned}$$

meaning each table captures the result of the corresponding function for all possible inputs. Figure 4 and Figure 5 show the table content. Looking at the entries, notice for example that

$$\begin{array}{lllllll} \text{ET}(255_{(10)}) & = & \text{ET}_{255_{(10)}} & = & 16_{(16)} & = & 22_{(10)} \\ \text{DT}(22_{(10)}) & = & \text{DT}_{22_{(10)}} & = & FF_{(16)} & = & 255_{(10)} \end{array}$$

i.e., the tables demonstrate the inverse nature of ET and DT. Beyond this, their descriptions give us two significant advantages:

1. Using the ET and DT functions implies extra computation; by using the ET-Tab and DT-Tab tables, we instead store all possible results for ET and DT then just look them up when we need to. There is no magic going on: there is just no point repeatedly invoking ET and DT given their result is fixed for a given input, and there is a fairly small number of inputs. This is a standard approach called **pre-computation**, i.e., "computation performed before we start": one one hand it make things more efficient in terms of time, but the trade-off is the space required to store the tables.

2. We can actually perform an example encryption and decryption. If we select

$$K = \langle 11_{(16)}, 22_{(16)} \rangle$$

and

$$P = \langle 33_{(16)}, 44_{(16)} \rangle$$

say, we can invoke Enc to perform an encryption

**Step** #1 Assign $P_0 \leftarrow \text{ET-Tab}_{P_0 \oplus K_0} = \text{ET-Tab}_{33_{(16)} \oplus 11_{(16)}} = 93_{(16)}$.
**Step** #2 Assign $P_1 \leftarrow \text{ET-Tab}_{P_1 \oplus K_1} = \text{ET-Tab}_{44_{(16)} \oplus 22_{(16)}} = 33_{(16)}$.
**Step** #3 Assign $t \leftarrow P_0 = 93_{(16)}$, $P_0 \leftarrow P_1 = 33_{(16)}$, $P_1 \leftarrow t = 93_{(16)}$.
**Step** #4 Assign $P_0 \leftarrow \text{ET-Tab}_{P_0 \oplus K_0} = \text{ET-Tab}_{33_{(16)} \oplus 11_{(16)}} = 93_{(16)}$.
**Step** #5 Assign $P_1 \leftarrow \text{ET-Tab}_{P_1 \oplus K_1} = \text{ET-Tab}_{93_{(16)} \oplus 22_{(16)}} = C8_{(16)}$.
**Step** #6 Assign $t \leftarrow P_0 = 93_{(16)}$, $P_0 \leftarrow P_1 = C8_{(16)}$, $P_1 \leftarrow t = 93_{(16)}$.
**Step** #7 Return $P = \langle C8_{(16)}, 93_{(16)} \rangle$.

that produces the ciphertext

$$\langle C8_{(16)}, 93_{(16)} \rangle.$$

Then, we can invoke Dec to perform the corresponding decryption

**Step** #1 Assign $t \leftarrow C_1 = 93_{(16)}$, $C_1 \leftarrow C_0 = C8_{(16)}$, $C_0 \leftarrow t = 93_{(16)}$.
**Step** #2 Assign $C_0 \leftarrow \text{DT-Tab}_{C_0} \oplus K_0 = \text{DT-Tab}_{93_{(16)}} \oplus 11_{(16)} = 33_{(16)}$.
**Step** #3 Assign $C_1 \leftarrow \text{DT-Tab}_{C_1} \oplus K_1 = \text{DT-Tab}_{C8_{(16)}} \oplus 22_{(16)} = 93_{(16)}$.
**Step** #4 Assign $t \leftarrow C_1 = 93_{(16)}$, $C_1 \leftarrow C_0 = 33_{(16)}$, $C_0 \leftarrow t = 93_{(16)}$.
**Step** #5 Assign $C_0 \leftarrow \text{DT-Tab}_{C_0} \oplus K_0 = \text{DT-Tab}_{93_{(16)}} \oplus 11_{(16)} = 11_{(16)}$.
**Step** #6 Assign $C_1 \leftarrow \text{DT-Tab}_{C_1} \oplus K_1 = \text{DT-Tab}_{33_{(16)}} \oplus 22_{(16)} = 22_{(16)}$.
**Step** #7 Return $C = \langle 11_{(16)}, 22_{(16)} \rangle$.

and recover the same plaintext as we started with, i.e.,

$$\langle 11_{(16)}, 22_{(16)} \rangle.$$

## 3.2 Implementing a simpler, straight-line version of the cipher

The example encryption and decryption above illustrate an important (and hopefully somewhat familiar) fact: once we select values for $n$ and $r$, the loops basically disappear and the example cipher just performs a sequence of assignments. Now we have settled on $n = 2$ and $r = 2$, we can apply the same approach as previously and write out straight-line versions of the Enc and Dec algorithms; the end result is shown in Figure 3.

Given that the algorithms are now so simple, we can actually go one better by translating the new straight-line version of Enc into a program for our example computer from Chapter 4. The implementation is shown in Figure 7 and is quite long in comparison to some others we have looked at. On the other hand, it is easy to explain step-by-step:

- Addresses #37 and #38 hold $K_0$ and $K_1$, (i.e., the key); addresses #40 and #41 hold $P_0$ and $P_1$, (i.e., the plaintext message). Addresses #43 onwards hold the entries of the S-box (i.e., the ET-Tab table) but to save space, these are not repeated: just imagine the content replicates Figure 4.

| Address | Instruction | Purpose |
|---|---|---|
| 0 | 220040 ↦ A ← MEM[40] | load $P_0$ |
| 1 | 320037 ↦ A ← A ⊕ MEM[37] | XOR $P_0$ and $K_0$ |
| 2 | 300042 ↦ A ← A + MEM[42] | compute "ET-Tab access" instruction |
| 3 | 210004 ↦ MEM[4] ← A | store "ET-Tab access" instruction |
| 4 | 000000 ↦ *NOP* | no operation |
| 5 | 210040 ↦ MEM[40] ← A | store $P_0$ |
| 6 | 220041 ↦ A ← MEM[41] | load $P_1$ |
| 7 | 320038 ↦ A ← A ⊕ MEM[38] | XOR $P_1$ and $K_1$ |
| 8 | 300042 ↦ A ← A + MEM[42] | compute "ET-Tab access" instruction |
| 9 | 210010 ↦ MEM[10] ← A | store "ET-Tab access" instruction |
| 10 | 000000 ↦ *NOP* | no operation |
| 11 | 210041 ↦ MEM[41] ← A | store $P_1$ |
| 12 | 220040 ↦ A ← MEM[40] | load $P_0$ |
| 13 | 210039 ↦ MEM[39] ← A | store $t$ |
| 14 | 220041 ↦ A ← MEM[41] | load $P_1$ |
| 15 | 210040 ↦ MEM[40] ← A | store $P_0$ |
| 16 | 220039 ↦ A ← MEM[39] | load $t$ |
| 17 | 210041 ↦ MEM[41] ← A | store $P_1$ |
| 18 | 220040 ↦ A ← MEM[40] | load $P_0$ |
| 19 | 320037 ↦ A ← A ⊕ MEM[37] | XOR $P_0$ and $K_0$ |
| 20 | 300042 ↦ A ← A + MEM[42] | compute "ET-Tab access" instruction |
| 21 | 210022 ↦ MEM[22] ← A | store "ET-Tab access" instruction |
| 22 | 000000 ↦ *NOP* | no operation |
| 23 | 210040 ↦ MEM[40] ← A | store $P_0$ |
| 24 | 220041 ↦ A ← MEM[41] | load $P_1$ |
| 25 | 320038 ↦ A ← A ⊕ MEM[38] | XOR $P_1$ and $K_1$ |
| 26 | 300042 ↦ A ← A + MEM[42] | compute "ET-Tab access" instruction |
| 27 | 210028 ↦ MEM[28] ← A | store "ET-Tab access" instruction |
| 28 | 000000 ↦ *NOP* | no operation |
| 29 | 210041 ↦ MEM[41] ← A | store $P_1$ |
| 30 | 220040 ↦ A ← MEM[40] | load $P_0$ |
| 31 | 210039 ↦ MEM[39] ← A | store $t$ |
| 32 | 220041 ↦ A ← MEM[41] | load $P_1$ |
| 33 | 210040 ↦ MEM[40] ← A | store $P_0$ |
| 34 | 220039 ↦ A ← MEM[39] | load $t$ |
| 35 | 210041 ↦ MEM[41] ← A | store $P_1$ |
| 36 | 100000 ↦ *HALT* | halt |
| 37 | 000017 ↦ *NOP* | $K_0$ |
| 38 | 000034 ↦ *NOP* | $K_1$ |
| 39 | 000000 ↦ *NOP* | $t$ |
| 40 | 000051 ↦ *NOP* | $P_0$ |
| 41 | 000068 ↦ *NOP* | $P_1$ |
| 42 | 220043 ↦ A ← MEM[43] | "ET-Tab access" instruction |
| 43 | 000099 ↦ *NOP* | ET-Tab$_0$, i.e., ET(0) |
| | ⋮ | |

**Figure 7:** *A partial (i.e., omitting most of the* ET-Tab *table) implementation of the* Enc *algorithm using the example computer.*

- The instructions in addresses #0 to #5 of mix together $K_0$ and $P_0$ the apply the substitution step using the S-box.

  This is actually quite tricky because we have no clear way to load values from the ET-Tab table. One neat solution is to make constructive use of self-modifying code from Chapter 4. Since the ET-Tab table starts at address #43, we *ideally* want to do something like

  $$A \leftarrow \mathsf{MEM}.$$

  This would allow us to compute $P_0 \oplus K_0$ into $A$, then use it as an offset from the start of the table to perform the look-up. But we cannot do this; there is no appropriate instruction. So instead, we "make" one ourselves: the idea is to take the value $P_0 \oplus K_0$ and add it to another value so as to produce the instruction we want as a result.

  This is better explained using an example. Imagine that

  $$P_0 \oplus K_0 = 51 \oplus 17 = 34$$

  and that we store this in $A$ via the instruction at address #1. Next we add the value at address #42, i.e., compute

  $$34 + 220043 = 220077 \mapsto A \leftarrow \mathsf{MEM}.$$

  So what we have is an instruction that loads from address #77 which just so happens to be the 34-th element of the ET-Tab table! We store this instruction at address #4, which is executed in the next step: we load $\mathsf{MEM} = 147$ and use the result to replace $P_0$.

- The instructions in addresses #6 to #11 do the same thing as addresses #0 to #5, but for $K_1$ and $P_1$ rather than $K_0$ and $P_0$.
- The instructions in addresses #12 to #17 implement the permutation step, swapping $P_0$ and $P_1$ using the temporary value $t$ which is held in address #39.
- The instructions in addresses #18 to #35 apply the second round, and basically just repeat addresses #0 to #17 again; address #36 is where execution halts.

The crucial thing to realise is that the program has 37 instructions it in; it takes 112 steps to execute on our example computer (a fetch, decode and execute step for each instruction plus one extra to reset it at the beginning) regardless of $K$ and $P$. However, some of those 37 instructions access memory. Remember that we are focusing on loads only: there are 22 in total including the ones we put there via self-modification. Based on what we have previously looked at, any variation in how long the program takes to execute is clearly down to how long these loads take.

Showing this in detail is a little laborious, but imagine we select

$$K = \langle 11_{(16)}, 22_{(16)} \rangle$$

and

$$P = \langle 33_{(16)}, 44_{(16)} \rangle$$

then execute the program on the example computer: it is equipped with the same cache as discussed previously, i.e., with $l = 8$ and $w = 4$ so the cache has eight lines, each of which can accommodate four sub-words. The 22 loads cause 16 cache-hits and 6 cache-misses in this case. If we use the same constants as previously, i.e.,

$$\begin{array}{lcl} T_{EXE} & = & 1 \\ T_{CACHE} & = & 5 \\ T_{MEM} & = & 10 \end{array}$$

and apply the same sort of estimation method, we get

$$\begin{array}{rcll} T & = & 37 \cdot T_{EXE} & + & 16 \cdot T_{CACHE} + 6 \cdot (T_{MEM} + T_{CACHE}) \\ & = & 37 \cdot 1 & + & 16 \cdot 5 + 6 \cdot (10 + 5) \\ & = & 207 & & \end{array}$$

as an estimate for the execution time. The point is that if we select

$$K = \langle 00_{(16)}, 00_{(16)} \rangle$$

and

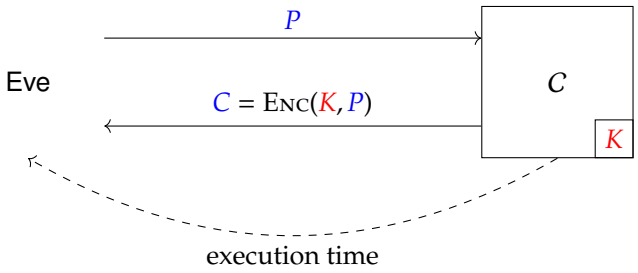
$$P = \langle 00_{(16)}, 00_{(16)} \rangle$$

instead, then the 22 loads cause cause 17 cache-hits and 5 cache-misses. This means the execution time is less:

$$\begin{array}{rcll} T & = & 37 \cdot T_{EXE} & + & 17 \cdot T_{CACHE} + 5 \cdot (T_{MEM} + T_{CACHE}) \\ & = & 37 \cdot 1 & + & 17 \cdot 5 + 5 \cdot (10 + 5) \\ & = & 197 & & \end{array}$$

# 4 A side-channel attack based on cache behaviour

Consider a similar problem to Chapter 12: some remote computer $C$ has a key $K$ embedded inside it, and our task as the attacker Eve is to guess $K$. We can model what is going on using a similar diagram as before:




So basically we send a plaintext message $P$ (which is public since we know it) to $C$, and get the ciphertext $C$ sent back; as well as $C$, we can measure how long the encryption takes. Of course the ET-Tab table is stored in the memory of $C$ so that it can apply the substitution step during encryption. When the cipher loads elements from the table, it does so using addresses derived from $K$ and $P$; as a result, the cache behaviour that results is partly dictated by $K$. Sometimes there will be lots of cache-hits, other times lots of cache-misses, but in either case the behaviour is in part due to $K$: if we time how long the cipher takes to execute, we learn something about $K$ because the only variation in execution time will because of the cache behaviour. Neat huh? Or maybe not so neat if it is *your* $K$ we are talking about.

## 4.1 A brute-force attack

$K$ is an $n$-element sequence: if each element, i.e., each $K_i$, is an 8-bit value then we know there must be $2^{8 \cdot n}$ possible values for $K$. As such, a brute-force attack would proceed as follows:

1. Pick a single random plaintext message $P$ (any one will do), and send it to $C$ to get the corresponding ciphertext
$$C = \text{Enc}(K, P).$$
2. For each possible key $K'$, compute
$$P' = \text{Dec}(K', C)$$
and then check whether $P' = P$. If the two match, we know that $K' = K$ and we have recovered the key.

The first step performs 1 encryption, while the second step performs $2^{8 \cdot 2} = 2^{16} = 65536$ encryptions in the worst case. Our goal therefore is to beat the brute-force attack by recovering $K$ using less than $1 + 65536 = 65537$ encryptions.

As an aside, you *could* argue that the second step is less important than the first: we can do this offline at our leisure rather than online, having to interact with $C$.

## 4.2 A side-channel attack

Putting together all the background material allows demonstration of a better attack that uses the side-channel information available to us: we have a cipher (with $n = 2$ and $r = 2$) implemented on a computer, and the computer is equipped with a cache (again with $l = 8$ and $w = 4$ as before). Look at the first statements in our straight-line version of the Enc algorithm in Figure 3 which correspond to the instructions in addresses #0 . . . #11 within the program in Figure 7. Basically we are interesting in this fragment

$$\begin{array}{l} \vdots \\ P_0 \leftarrow \text{ET-Tab}_{P_0 \oplus K_0} \\ P_1 \leftarrow \text{ET-Tab}_{P_1 \oplus K_1} \\ \vdots \end{array}$$

which includes two accesses to the ET-Tab table. If the second access causes a cache-hit, we know that the address used (i.e., the index into the ET-Tab table) must match the address used by the first access: this is what a cache-hit means. So essentially we are allowed to write something like

$$P_0 \oplus K_0 = P_1 \oplus K_1.$$

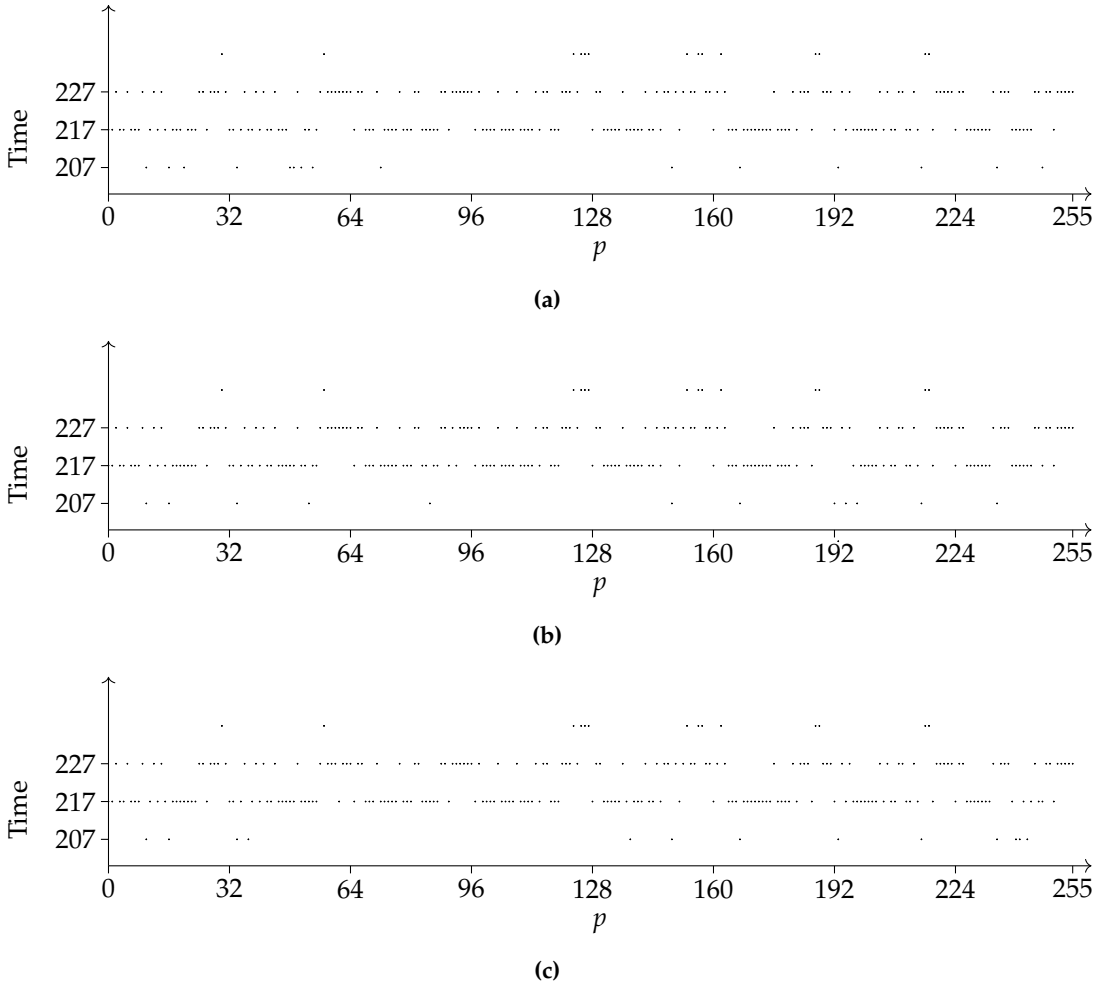



**Figure 8:** *Graphs showing variation in the execution time caused by difference plaintexts of the form* $P = \langle f, p \rangle$ *for fixed* $f$ *and variable* $p$.

But this is not quite true: while the addresses *match* according to the rules of the cache, they are not *equal*. Why not? Remember there is more than one sub-word in each cache line; a match only means the addresses map to the same cache line not that they access the same sub-word. Another way to say the same thing is that the addresses *are* equal *if* we compare the "sub-wordless" versions. So instead we write

$$P_0 \oplus K_0 \equiv P_1 \oplus K_1$$

meaning the left-hand side is equivalent to the right-hand side if we convert both into their sub-wordless form. Even so, this is quite useful: we *know* $P_0$ and $P_1$, since we decided on their value. In reality this means we can write

$$P_0 \oplus P_1 \equiv K_0 \oplus K_1$$

where we know the left-hand side but not the right-hand side. Technically speaking, this is a **difference** between $K_0$ and $K_1$: we know neither $K_0$ nor $K_1$, but do know a relationship between the two. Put another way, if we guess $K_0$ then we can compute

$$K_1 \equiv P_0 \oplus P_1 \oplus K_0$$

without having to guess this part of the key independently

Of course the challenge is now to first acquire and then capitalise on this difference. Imagine that

$$K = \langle 11_{(16)}, 22_{(16)} \rangle$$

but remember that as the attacker we obviously do not know this. The attack consists of two phases which are described below.

#### 4.2.1 Phase #1: acquisition

In the first phase, we try to acquire the difference (or at least a small set of candidates).

1. Set $f = 0$ and send every special plaintext message of the form

   $$P = \langle f, p \rangle$$

   for every possible value $p \in \{0, 1, \ldots, 255\}$, to $C$ We throw away the corresponding ciphertexts but capture the execution time and plot it as the graph; this one is shown at the top of Figure 8.

   The plot illustrates an interesting feature: some choices of $P$ (i.e., choices of $p$) mean a lower execution time than others. Most have an execution time of 217, but a few, i.e., those $p$ in the set

   $$A = \{10, 16, 20, 34, 48, 49, 51, 54, 72, 149, 167, 193, 215, 235, 247\},$$

   have a particularly low execution time of 207. Each $p$ gives us a candidate difference: remember we can write

   $$P_0 \oplus P_1 \equiv K_0 \oplus K_1$$

   because if the execution time is low, there are many cache-hits. If we fill in what we know, we can calculate each difference as

   $$f \oplus p \equiv K_0 \oplus K_1.$$

   But remember we only really care about the sub-wordless versions of these differences, so actually we can can transform $A$ into

   $$A_{sub-wordless} = \{8, 16, 20, 24, 32, 48, 52, 72, 148, 164, 192, 212, 232, 244\}.$$

2. Repeat the first step again, but this time select $f = 240$; this yields another set

   $$B = \{10, 16, 34, 53, 85, 149, 167, 192, 195, 198, 215, 235\}$$

   from the middle graph in Figure 8, and hence

   $$B_{sub-wordless} = \{24, 36, 48, 52, 84, 100, 164, 196, 208, 224, 248\}.$$

3. Repeat the first step again, but this time select $f = 192$; this yields another set

   $$C = \{10, 16, 34, 37, 138, 149, 167, 193, 215, 235, 240, 241, 243\}$$

   from the bottom graph in Figure 8, and hence

   $$C_{sub-wordless} = \{0, 20, 40, 48, 72, 84, 100, 200, 208, 224, 228\}.$$

Finally the magic happens: notice that

$$D = A_{sub-wordless} \cap B_{sub-wordless} \cap C_{sub-wordless} = \{48\}$$

meaning that a single difference, i.e.,

$$48 \equiv K_0 \oplus K_1$$

agrees in all three cases. Sometimes this does not work out so well and instead of a single difference we are forced to work with a *set* of candidates. This does not matter too much: the next phase is the same if we have one candidate or many, having one just means we do less work.

#### 4.2.2 Phase #2: analysis

Armed with $D$ (which for us is just a single candidate difference) we can now perform a "reduced" brute-force search that will recover $K$ for us:

1. Pick a single random plaintext message $P$ (any one will do), and send it to $C$ to get the corresponding ciphertext
$$C = \text{Enc}(K, P).$$
2. For every special key of the form
$$K' = \langle k, (k \oplus d) + i \rangle$$
for $k \in \{0, 1, \ldots, 255\}$, $d \in D$ and $i \in \{0, 1, \ldots, w - 1\}$, compute
$$P' = \text{Dec}(K', C)$$
and then check whether $P' = P$. If the two match, we know that $K' = K$ and we have recovered the key. For our example, we succeed when $k = 17$, $d = 48$ and $i = 1$ meaning that
$$K' = \langle 17, (17 \oplus 48) + 1 \rangle = \langle 17, 34 \rangle = K.$$
Notice that the expression $(k \oplus d) + i$ is a bit like what we did in the Load algorithm when fetching a cache line from MEM: basically we have a sub-wordless address computed by fixing up $k$ using the difference $d$ being considered, and then cycle through all the addresses in the cache line $k \oplus d$ maps into.

#### 4.2.3 The end result

How many encryptions do we perform? Consider the two phases:

- The acquisition phase timed the execution of $2^8 = 256$ special plaintexts, repeating the process three times and thus performing $3 \cdot 256 = 768$ encryptions.
- The analysis phase is somewhat similar to the brute-force attack: the first step performs 1 encryption, while since we had just one difference in $D$ the second step performs $256 \cdot 1 \cdot w = 1024$ encryptions in the worst case.

In total, the side-channel attack therefore performed $768 + 1 + 1024 = 1793$ encryptions: this is significantly less than the 65537 performed by the brute-force attack. On the other hand, this time we needed $1 + 768$ online encryptions rather than 1, and so need to interact with $C$ more often.

If you look back at all the simplifications we have made (e.g., the cache design, the cipher design etc.) there is a temptation to say "so what". The point is that although this is a simplification, *real* attacks on AES exist and really do follow similar reasoning: they *are* more complicated, but they also start to extract a bigger advantage. Remember that very roughly, AES use $n = 16$. A brute-force attack would therefore perform an unfeasibly large

$$2^{8 \cdot 16} = 2^{128} = 340282366920938463463374607431768211456$$

encryptions to recover $K$. Modern cache based side-channel attacks on AES perform somewhat less than

$$2^{32} = 4294967296$$

encryptions: this is *very* feasible, and hence a much more clear threat.


In Chapter 12, we explored a range of countermeasures designed to prevent (with varying degrees of success) a side-channel attack based on leakage of execution time from Match-Pwd. Of course we could do a similar thing here: can you think of how similar concepts could be applied to our block cipher Enc (keeping in mind that the *combination* of program *and* computer, or cache at least, are now to blame)?

## References

[1] *Wikipedia: Advanced Encryption Standard (AES)*. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard (see p. 17).

[2] *Wikipedia: Block cipher*. http://en.wikipedia.org/wiki/Block_cipher (see p. 17).

[3] *Wikipedia: Cache*. http://en.wikipedia.org/wiki/Cache (see p. 3).

[4] *Wikipedia: Inverse function*. http://en.wikipedia.org/wiki/Inverse_function (see p. 17).

[5] *Wikipedia: Memory hierarchy*. http://en.wikipedia.org/wiki/Memory_hierarchy (see p. 3).

[6] *Wikipedia: S-box*. http://en.wikipedia.org/wiki/S-box (see p. 17).

[7] *Wikipedia: Substitution-Permutation network*. http://en.wikipedia.org/wiki/Substitution-permutation_network (see p. 17).

[8] *Wikipedia: The magical number seven, plus or minus two*. http://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two (see p. 4).

**I have a question/comment/complaint for you.** Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way.

However, keep in mind we are far from perfect; mistakes are of course possible, although hopefully rare. Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

**Can I use this material for something ?** We are distributing these PDFs under the Creative Commons Attribution-ShareAlike License (CC BY-SA)

http://creativecommons.org/licenses/by-sa/4.0/

This is a fancy way to say you can basically do whatever you want with them (i.e., use, copy and distribute it however you see fit) as long as a) you correctly attribute the original source, and b) you distribute the result under the same license.

**Is there a printed version of this material I can buy?** Yes: Springer have published selected Chapters in

http://www.springer.com/computer/book/978-3-319-04041-7

while *also* allowing us to keep electronic versions online. Any royalties from this published version are donated to the UK-based Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

**Why are all your references to Wikipedia?** Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

**I like programming; why do the examples include so little programming?** We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

**But you need to be able to program to do Computer Science, right?** Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.