What is Computer Science?

An Information Security Perspective

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

git # 148def3 @ 2018-07-11

PLAYING HIDE-AND-SEEK WITH VIRUS SCANNERS

A **computer virus** [5] is a program that attempts to **infect** another **host** file (often another program) by replicating itself (e.g., appending a copy of itself to the host). This is analogous to how Biological viruses attach themselves to host cells. The idea is that when the host file is loaded or executed, the virus is activated: at this point the virus can deploy a **payload** of some kind, which often has malicious intent. Although a definition[1] for computer viruses was yet to be written in 1971, it represents the point in time when instances of programs we retrospectively call viruses started to appear. **Creeper** [7], written by Bob Thomas, is often cited as the first: it targeted PDP [19] computers connected to the **Advanced Research Projects Agency Network (ARPANET)** [3], propagating itself via the telephone system. Once resident it displayed the annoying but otherwise non-destructive message "I'm the creeper, catch me if you can" to users, but seemingly did *not* replicate itself (at least intentionally): once the payload was deployed, it simply moved on to the next computer. Perhaps more (in)famous though, is the 1988 **Morris Worm** [17] written by Robert Morris. Although it again lacked a malign payload, the worm infected computers multiple times, slowing down their normal operation to the point where roughly 10% of computers connected to the **Internet** [14] were claimed to have been left non-operational.

A rich published history [24] of computer viruses now exists; since much of it is legally dubious, more undoubtedly remains unpublished. Both Creeper and the Morris Worm represented an experimental era in this historical timeline, with honest security researchers exploring what was and was not possible; neither was intentionally malicious, with Creeper being explicitly described as an experimental **self-replicating** program. Beyond the technological aspects, the Morris Worm is also notorious because of the subsequent prosecution of Morris and treatment of information, network and computer security as an important emerging threat: governments, rather than just programmers and researchers, started to take notice. Today of course, viruses (and **malware** [16] more generally) have become big business for the wider computer underground. Not content with sending you annoying messages, a modern virus might delete or encrypt your files (holding the owners to ransom, so they must pay before the content can be read again), steal information [15], or even attempt to actively destroy *physical* infrastructure. **Stuxnet** [23], for instance, represents what is widely viewed as a "cyber-weapon" developed and deployed by some (unnamed) government against uranium enrichment plants in Iran.

In many cases therefore, (cyber-)crime increasingly *does* pay. The result is an arms race between the people who write viruses, and the people who write **anti-virus** software. Again drawing on Biology (and the field of virotherapy), the first example of anti-virus was arguably a second virus called **Reaper** written to stamp out the spread of updates to **Creeper** (which, in contrast to the original, *did* replicate themselves). More generally however, before we can disable a virus we need to detect whether a given host file is infected or not. Interestingly we can *prove* this is a good business to be in by showing that it is *impossible* to write a perfect virus detector algorithm. Imagine for a moment we *could* do this: DETECT represents a perfect virus detector, so we are sure DETECT(x) = **true** whenever x is a virus, and that DETECT(x) = **false** in all other cases. Now imagine

[1] Strictly speaking, a **virus** is a program that propagates itself from file to file on one computer, but typically requires an external stimulus to propagate between computers (e.g., a user carrying infected files on a USB stick from one computer to another); the requirement for a host file to infect means the virus is typically not a stand-alone program. This contrasts with a **worm**, which propagates from computer to computer itself, acting as a stand-alone program without the need to infect a host file. A specific example might include aspects of both, so a precise classification is often difficult; we largely ignore the issue, using the term virus as an imprecise but convenient catch-all.

there is a virus called Virus. It is clever: it incorporates a *copy* of Detect, the virus detector algorithm, inside itself. A simple description of how Virus works is as follows:

```
1 algorithm Virus begin
2 |  if Detect(Virus) = true then
3 |  |  Behave like a non-virus, i.e., a normal program
4 |  else
5 |  |  Behave like a virus, i.e., do something "bad"
6 |  end
7 end
```

So when Virus is executed, it checks whether Detect says it is a virus. If Detect(Virus) = **true**, i.e., Detect thinks Virus is a virus, then Virus does nothing. Hence, in this case Virus *does not* behave like a virus and so Detect was wrong. If Detect(Virus) returns **false**, i.e., Detect does not think Virus is a virus, then Virus does something "bad" such as deleting all your files. In this case Virus definitely *does* behave like a virus, and so Detect is wrong again. Thus, if we assume a perfect virus scanner Detect exists then we get a contradiction; this means Detect cannot exist. Although this proof relates to writing perfect anti-virus software, it is in fact a special case of the famous **Halting problem** [10].

Despite the fact that we cannot write perfect anti-virus software, it is still worth looking at how we might at least *try* to detect V; a common tool in this context is a virus **scanner** [2]. Most work in roughly the same way: the idea is that for each virus, one tries to create a unique **signature** that identifies files containing the virus. The scanner inspects each file in turn, and tries to match the file content with each signature. If there are no matches, then the scanner concludes that there is no virus in the file and moves on to the next one. We can imagine both the signature and the file being sequences of numbers called S and F. There are lots of efficient ways to search F, the easiest of which is to simply take each S and match it up against the content of F one element at a time:

$$
\begin{array}{rcccccccccccccccc}
F & = & \langle & \ldots, & 6, & 5, & 2, & 1, & 4, & 0, & 3, & 5, & 2, & 1, & \ldots & \rangle & & \\
S & = & \langle & & 3, & 5, & 2, & 1 & & & & & & & & \rangle & \mapsto & \textbf{no match} \\
 & & \langle & & & 3, & 5, & 2, & 1 & & & & & & & \rangle & \mapsto & \textbf{no match} \\
 & & \langle & & & & 3, & 5, & 2, & 1 & & & & & & \rangle & \mapsto & \textbf{no match} \\
 & & \langle & & & & & 3, & 5, & 2, & 1 & & & & & \rangle & \mapsto & \textbf{no match} \\
 & & \langle & & & & & & 3, & 5, & 2, & 1 & & & & \rangle & \mapsto & \textbf{no match} \\
 & & \langle & & & & & & & 3, & 5, & 2, & 1 & & & \rangle & \mapsto & \textbf{no match} \\
 & & \langle & & & & & & & & 3, & 5, & 2, & 1 & & \rangle & \mapsto & \textbf{match}
\end{array}
$$

In the i-th step, we compare S against the sub-sequence of F starting at F_i. We continue this process until either we run out of content, or there is a match between S and a particular region of F. In the latter case, we have found an occurrence of S in F and identified the file as containing the virus. This approach relies on the fact that the people who sold you the virus scanner send you a new signature whenever a new virus is detected somewhere; this is basically why you need to update your virus scanner regularly. A given signature might not be perfect, and might sound the alarm for some files that *do not* contain a virus. For example, the signature $\langle 3, 5, 2, 1 \rangle$ might be a valid part of some uninfected data file, but would still cause the scanner to flag it as infected.

In this Chapter we will play the part of the bad guy, and imagine we want to write a virus. The goal is to explain how our virus can avoid detection by virus scanners (or at least the one described above), and yet still execute some payload. The approach we use is to have the virus change itself *during* execution so as to fool the virus scanner. This sort of self-modifying program is made possible by the way that modern computers execute programs. So before we start talking about viruses, we will start by looking at what programs are and how computers execute them.

1 Computers and programs

The questions "what is a computer" and "what is computation" are at the core of Computer Science; eminent scholars in the field have spent, and still spend, lots of time thinking about and producing new results in this area. If you pick up a textbook on the subject, the topic of **Turing Machines (TMs)** [25] is often used as a starting point. Alan Turing [1], some might say the father of Computer Science, introduced TMs as a theoretical tool to reason about computers and computation: we still use them for the same tasks today. But TMs and their use are a topic in their own right, and one not quite aligned to what we want here. Specifically, we would like to answer some more concrete, more practical questions instead. So, consider that you probably sit in front of and use a physical (rather than theoretical) computer every day: we would like to know "what is *that* computer" and more importantly "how does *it* compute things"?

1.1 A theoretical computer

A computer is basically just a machine used to process steps, which we call **instructions**, collected together into **programs**. This should sound familiar: the computer is doing something similar to what *we* do when we step through an algorithm. In a sense, the only thing that makes a computer remarkable is that it **executes** the instructions in a program *much* faster than we can process algorithms, and more or less without error. Just like we can process any reasonably written algorithm, a computer can execute any program. This is a neat feature: we do not need one computer to send emails and a different one to view web-pages, we just need one **general-purpose** computer that can do more or less anything when provided with an appropriate program.

To design a computer, we need to write down an algorithm that describes how to process programs (which are simply algorithms). In a very rough sense, as the user of a computer you "see" the result as the combination of an **operating system** (e.g., UNIX) and a **Central Processing Unit (CPU)** [4], or **processor**, that is the main hardware component within the computer. Basically the processor is the thing that actually executes programs, and the operating systems acts as an assistant by loading the programs from disk, allowing you to select which program to execute and so on. Of course we then need to build the hardware somehow, but we will worry about that later; basically what we want as a starting point is an algorithm for processing algorithms. Here is a first attempt:

1. Write a program as a sequence of instructions called P; start executing the sequence from the first element, i.e., P_0.
2. Fetch the next instruction from the program and call it IR.
3. If $IR = \perp$ (i.e., we have run out of instructions to execute) or if $IR = HALT$ then halt the computer, otherwise execute IR (i.e., perform the operation it specifies).
4. Repeat from line #2.

Look at this a little more closely. The sequence of instructions we call P is a program for the computer; if it makes things easier, think of it as a word processor or web-browser or something. Each element of P is an instruction, and each instruction is taken from an **instruction set** of possibilities that the computer understands. Lines #2 to #4 form a loop. During each iteration we first fetch the next instruction from P. We call the instruction IR because in real computers, it is held in the **Instruction Register (IR)**. Once we have IR to hand, we set about performing whatever operation is specifies; then if we encounter a special instruction called $HALT$ we stop execution, otherwise we carry out the whole loop again.

Hopefully this seems sensible. It should do, because as we have tried to motivate, it more or less models the same thing *you* would do if you were processing the steps of an algorithm. An example makes the whole idea easier to understand: imagine we want to execute a short program that computes $10 + 20$. First we write down the program, e.g.,

$$P = \langle A \leftarrow 10, A \leftarrow A + 20, HALT \rangle,$$

and then process the remaining steps of our algorithm to execute it

Step #1 Fetch $IR = A \leftarrow 10$ from the sequence.

Step #2 Since $IR \neq \perp$ and $IR \neq HALT$, execute $A \leftarrow 10$, i.e., set A to 10.

Step #3 Fetch $IR = A \leftarrow A + 20$ from the sequence.

Step #4 Since $IR \neq \perp$ and $IR \neq HALT$, execute $A \leftarrow A + 20$, i.e., set A to 30.

Step #5 Fetch $IR = HALT$ from the sequence.

Step #6 Since $IR \neq \perp$ but $IR = HALT$, halt the computer.

after which we have the result $10 + 20 = 30$ held in A. Hopefully the underlying point is clear. Specifically, bar the issue with building the hardware itself there is no magic behind the scenes: this *really is* how a computer executes a program.

1.2 A real, Harvard-style computer

It turns out that the hardware components we would need to build a computer like the one described above relate well to those you would find in a simple pocket calculator. For example:

A pocket calculator:

- Has an accumulator (i.e., the current value).
- Has some memory (accessed via the *M*+ and *MR* buttons) that can store values.
- Has some device to perform arithmetic (i.e., to add together numbers).
- Has input and output peripherals (e.g., keypad, LCD screen).
- Responds to simple commands or instructions from the user; for example the user can supply things to perform arithmetic on (i.e., numbers) and commands to perform arithmetic (e.g., do an addition).

A computer:

- Has many accumulators (often called registers).
- Has potentially many levels and large amounts of memory (often called RAM).
- Has an Arithmetic and Logic Unit (ALU) to perform arithmetic.
- Has input and output peripherals (e.g., keyboard, mouse, hard disk, monitor).
- Executes sequences of simple instructions called programs; each instruction consists of some **operands** (i.e., the things to operate on) and an **opcode** (i.e., the operation to perform).

In fact, early computers essentially *were* very large versions of what today we would call a calculator. They were used to perform repetitive computation, such as computing tables of Sin and Cos for people to use. Computers of this era typically relied on

1. a paper **tape** [20], where instructions from the program were stored as patterns of holes in the paper, and
2. some **memory**, where data being processed by the computer was stored.

We could expand on both technologies, but their detail is not really that important. Instead we can model them using two sequences called TAPE and MEM: MEM[i] represents the i-th **address** in memory, whereas TAPE[j] represents the j-th row on the continuous reel of paper tape.

So imagine we want to build an example computer of this type. We assume it has a tape and a memory as described above, and single **accumulator** called A. Each element of TAPE holds an instruction from the program, while each element of MEM and the accumulator A can hold numbers; to make things simpler, we will write each number using decimal. The computer can understand a limited set of instructions:

- `NOP`, i.e., do nothing.
- `HALT`, i.e., halt or stop execution.
- A $\leftarrow n$, i.e., load the number n into the accumulator A.
- MEM[n] $\leftarrow$ A, i.e., store the number in accumulator A into address n of the memory.
- A $\leftarrow$ MEM[n], i.e., load the number in address n in memory into the accumulator A.
- A $\leftarrow$ A + MEM[n], i.e., add the number in address n of the memory to the accumulator A and store the result back in the accumulator.
- A $\leftarrow$ A – MEM[n], i.e., subtract the number in address n of the memory from the accumulator A and store the result back in the accumulator.
- A $\leftarrow$ A $\oplus$ MEM[n], i.e., XOR the number in address n of the memory with the accumulator A and store the result back in the accumulator.

One can view things on the right-hand side of the assignment symbol $\leftarrow$ as being read from, and those on the left-hand side as being written to by an instruction. So for example, an instruction

$$A \leftarrow 10$$

reads the number 10 and writes it into A. This also implies that memory access can be written in the same way; for example

$$\mathsf{MEM}[64] \leftarrow A$$

reads the number in A and writes it into memory at address sixty four.

It is important to notice that a given program for our example computer can *only* include instructions from this instruction set. It simply does not know how to execute any other type. For example, we could not feed it the Fermat-Test algorithm from Chapter 3 because it uses operations not included in the instruction set. Even

CPU		
state	=	reset
IR	=	
A	=	0

MEM	
Address	*Value*
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	*Meaning*
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	*HALT*
4	*NOP*
5	*NOP*
6	*NOP*
7	*NOP*

(a) *Step* #1*: Load the tape into the tape reader and start the computer.*

CPU		
state	=	fetch
IR	=	A ← MEM[4]
A	=	0

MEM	
Address	*Value*
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	*Meaning*
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	*HALT*
4	*NOP*
5	*NOP*
6	*NOP*
7	*NOP*

(b) *Step* #2*: Fetch the next instruction* IR = A ← MEM[4] *from the tape.*

CPU		
state	=	execute
IR	=	A ← MEM[4]
A	=	10

MEM	
Address	*Value*
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	*Meaning*
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	*HALT*
4	*NOP*
5	*NOP*
6	*NOP*
7	*NOP*

(c) *Step* #3*: Execute the instruction* A ← MEM[4]*, i.e., set* A *to* MEM[4] = 10*.*

CPU		
state	=	fetch
IR	=	A ← A + MEM[5]
A	=	10

MEM	
Address	*Value*
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	*Meaning*
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	*HALT*
4	*NOP*
5	*NOP*
6	*NOP*
7	*NOP*

(d) *Step* #4*: Fetch the next instruction* IR = A ← A + MEM[5] *from the tape.*

CPU		
state	=	execute
IR	=	A ← A + MEM[5]
A	=	30

MEM	
Address	*Value*
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	*Meaning*
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	*HALT*
4	*NOP*
5	*NOP*
6	*NOP*
7	*NOP*

(e) *Step* #5*: Execute the instruction* A ← A + MEM[5]*, i.e., set* A *to* A + MEM[5] = 30*.*

CPU		
state	=	fetch
IR	=	MEM[6] ← A
A	=	30

MEM	
Address	*Value*
0	0
1	0
2	0
3	0
4	10
5	20
6	0
7	0

TAPE	
Address	*Meaning*
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	*HALT*
4	*NOP*
5	*NOP*
6	*NOP*
7	*NOP*

(f) *Step* #6*: Fetch the next instruction* IR = MEM[6] ← A *from the tape.*

Figure 1: *Computing the sum* 10 + 20*, as executed on a Harvard-style computer.*

CPU		
state	=	execute
IR	=	MEM[6] ← A
A	=	30

MEM	
Address	*Value*
0	0
1	0
2	0
3	0
4	10
5	20
6	30
7	0

TAPE	
Address	*Meaning*
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	*HALT*
4	*NOP*
5	*NOP*
6	*NOP*
7	*NOP*

(a) *Step #7: Execute the instruction* MEM[6] ← A*, i.e., set* MEM[6] *to* A = 30*.*

CPU		
state	=	fetch
IR	=	*HALT*
A	=	30

MEM	
Address	*Value*
0	0
1	0
2	0
3	0
4	10
5	20
6	30
7	0

TAPE	
Address	*Meaning*
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	*HALT*
4	*NOP*
5	*NOP*
6	*NOP*
7	*NOP*

(b) *Step #8: Fetch the next instruction* IR = *HALT from the tape.*

CPU		
state	=	execute
IR	=	*HALT*
A	=	30

MEM	
Address	*Value*
0	0
1	0
2	0
3	0
4	10
5	20
6	30
7	0

TAPE	
Address	*Meaning*
0	A ← MEM[4]
1	A ← A + MEM[5]
2	MEM[6] ← A
3	*HALT*
4	*NOP*
5	*NOP*
6	*NOP*
7	*NOP*

(c) *Step #9: Execute the instruction HALT, i.e., stop execution.*

Figure 2: *Computing the sum* 10 + 20*, as executed on a Harvard-style computer.*

so, we *can* start to write useful programs. Consider a similar example to the one we looked at previously: we want to add together two numbers held in MEM[4] and MEM[5] (where say MEM[4] = 10 and MEM[5] = 20), and then store the result into MEM[6]. The program consists of the four instructions

$$
\begin{aligned}
&A \leftarrow \mathsf{MEM}[4] \\
&A \leftarrow A + \mathsf{MEM}[5] \\
&\mathsf{MEM}[6] \leftarrow A \\
&\mathit{HALT}
\end{aligned}
$$

The method used by the computer to execute a program like this is *very* similar to our first attempt above:

1. Encode a program P onto paper tape; load this tape into the tape reader, then start the computer.
2. Using the tape reader, fetch the next instruction in the program and call it IR.
3. If
   (a) IR $=\perp$ (i.e., an invalid instruction is encountered) or
   (b) IR = `HALT` (i.e., the program concludes normally)

   then halt the computer, otherwise execute IR (i.e., perform the operation it specifies).
4. Repeat from line 2.

We can describe each step during execution of the program by detailing the state of the computer, e.g., what values are held by MEM, TAPE and A; Figure 1 and Figure 2 do just this. The execution shows that after step #9, the computer halts and we end up with the result of the addition, i.e., 30, stored in MEM[6]; not exactly a word processor or web-browser, but quite an achievement by the standards of the 1940s!

It is important to remember that the program is free to alter memory content, but has no means of altering the tape which houses the instructions. That is, once we start the computer, we cannot change the program: our example computer views instructions and the data as fundamentally *different* things. This is now termed a **Harvard architecture** [11] after the **Automatic Sequence Controlled Calculator (ASCC)** designed by Howard Aiken and built by IBM; the installation at Harvard University, delivered in around 1944, was nicknamed the "Mark 1" [12]. The crucial thing to take away is that there is still no magic involved: as shown in Figure 5.1, the Harvard Mark 1 was a real, physical computer built on *exactly* the principles as above.

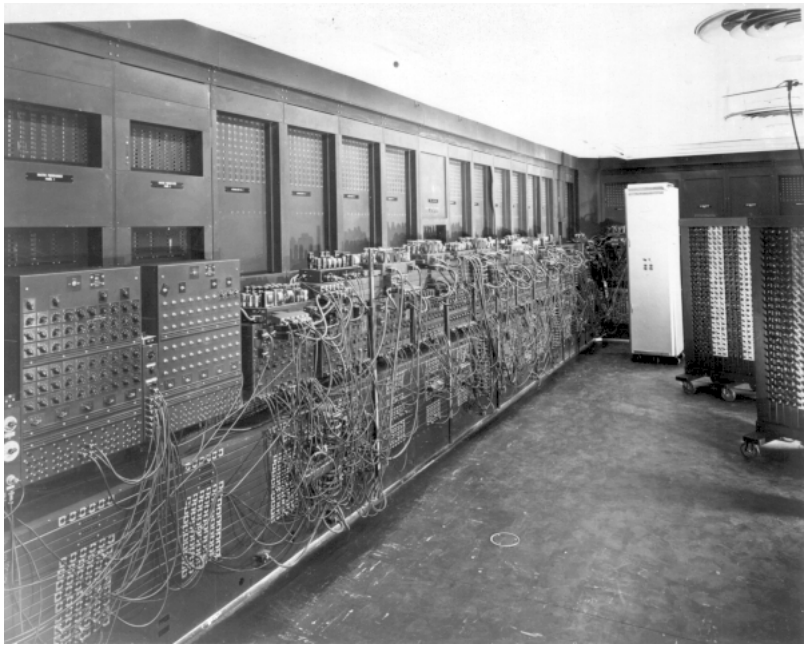

Figure 3: *The ENIAC installation at the US Army Ballistics Research Laboratory (public domain image, source: US Army Photo* `http://ftp.arl.army.mil/ftp/historic-computers/gif/eniac5.gif`*).*

1.3 A real, von Neumann-style computer

The Harvard style view of computers changed radically when **John von Neumann** [26] documented the concept of a **stored program architecture** in around 1945. The basic idea is that instructions and data are actually the *same* thing. Think about it: what does 1 mean? The meaning of 1 completely depends on the context it is placed in, and how *we* interpret it: if we interpret it as a number it means the integer one, if we interpret it as an instruction it could mean "perform an addition". So basically, as long as we have an encoding from numbers to meaning then we can store *both* instructions and data as numbers in memory. For our purposes, the encoding does not matter too much; imagine we continue to represent everything as decimal numbers:

- 00*nnnn* means `NOP`.
- 10*nnnn* means `HALT`.
- 20*nnnn* means $\mathsf{A} \leftarrow n$.
- 21*nnnn* means $\mathsf{MEM}[n] \leftarrow \mathsf{A}$.
- 22*nnnn* means $\mathsf{A} \leftarrow \mathsf{MEM}[n]$.
- 30*nnnn* means $\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[n]$.
- 31*nnnn* means $\mathsf{A} \leftarrow \mathsf{A} - \mathsf{MEM}[n]$.
- 32*nnnn* means $\mathsf{A} \leftarrow \mathsf{A} \oplus \mathsf{MEM}[n]$.
- 40*nnnn* means $\mathsf{PC} \leftarrow n$.
- 41*nnnn* means $\mathsf{PC} \leftarrow n$ iff. $\mathsf{A} = 0$.
- 42*nnnn* means $\mathsf{PC} \leftarrow n$ iff. $\mathsf{A} \neq 0$.

The entries on the left-hand side perhaps need some explanation. Take the entry for 20*nnnn*: this is a six digit decimal number where the left-most two digits are 2 and 0, and the right-most four can be *any* digits in the range $0, 1, \ldots, 9$. On the right-hand side, we replace *nnnn* by a single number n so it will be in the range $0, 1, \ldots, 9999$. For example, the decimal number 300005, viewed as an encoded instruction 300005, means $A \leftarrow A + \mathsf{MEM}$: we match 0005 on the left-hand side of the table, and turn it into 5 on the right-hand side. The 30 part is the **opcode**, and the 0005 part is the **operand**; the 30 part tells the computer what operation to perform, while the 0005 part tells it what to perform the operation on.

So if we store both instructions and data in memory, how do we know which instruction to execute next? Basically, we need an extra accumulator, which we call the **Program Counter (PC)**, to keep track of where to

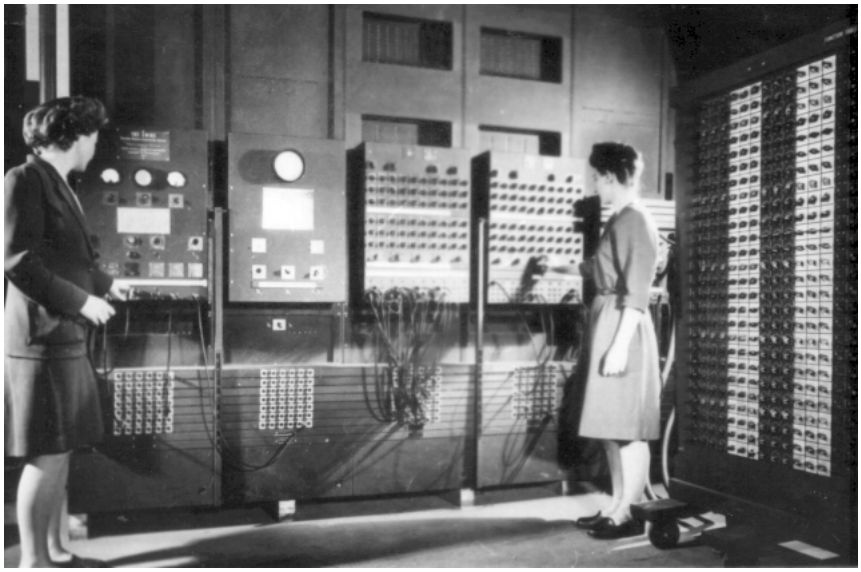

Figure 4: *Two operators at the ENIAC control panel (public domain image, source: US Army Photo http://ftp.arl.army.mil/ftp/historic-computers/gif/eniac7.gif).*

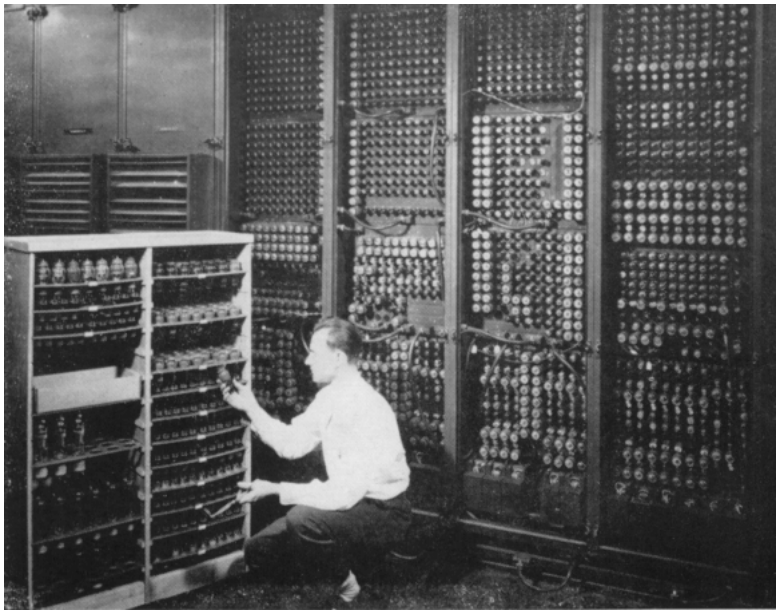

Figure 5: *Original caption notes that "replacing a bad [vacuum] tube meant checking among ENIACs 19,000 possibilities"; a daunting task! (public domain image, source: US Army Photo http://ftp.arl.army.mil/ftp/historic-computers/gif/eniac3.gif).*

CPU		
state	=	reset
PC	=	0
IR	=	
	=	
A	=	0

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	0	*NOP*
7	0	*NOP*

(a) *Step* #1*: Load the tape into memory, set* PC = 0 *and start the computer.*

CPU		
state	=	fetch
PC	=	0
IR	=	220004
	=	
A	=	0

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	0	*NOP*
7	0	*NOP*

(b) *Step* #2*: Fetch the next instruction* IR = 220004 *from* PC = 0*.*

CPU		
state	=	decode
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	0

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	0	*NOP*
7	0	*NOP*

(c) *Step* #3*: Decode* IR = 220004 *into* A ← MEM[4]*, and set* PC *to* PC + 1 = 1*.*

CPU		
state	=	execute
PC	=	1
IR	=	220004
	=	A ← MEM[4]
A	=	10

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	0	*NOP*
7	0	*NOP*

(d) *Step* #4*: Execute the instruction* A ← MEM[4]*, i.e., set* A *to* MEM[4] = 10*.*

CPU		
state	=	fetch
PC	=	1
IR	=	300005
	=	
A	=	10

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	0	*NOP*
7	0	*NOP*

(e) *Step* #5*: Fetch the next instruction* IR = 300005 *from* PC = 1*.*

CPU		
state	=	decode
PC	=	2
IR	=	300005
	=	A ← A + MEM[5]
A	=	10

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	0	*NOP*
7	0	*NOP*

(f) *Step* #6*: Decode* IR = 300005 *into* A ← A + MEM[5]*, and set* PC *to* PC + 1 = 2*.*

Figure 6: *Computing the sum* 10 + 20*, as executed on a stored program computer.*

CPU		
state	=	execute
PC	=	2
IR	=	300005
	=	A ← A + MEM[5]
A	=	30

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	0	*NOP*
7	0	*NOP*

(a) *Step #7: Execute the instruction* A ← A + MEM[5]*, i.e., set* A *to* A + MEM[5] = 30.

CPU		
state	=	fetch
PC	=	2
IR	=	210006
	=	
A	=	30

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	0	*NOP*
7	0	*NOP*

(b) *Step #8: Fetch the next instruction* IR = 210006 *from* PC = 2.

CPU		
state	=	decode
PC	=	3
IR	=	210006
	=	MEM[6] ← A
A	=	30

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	0	*NOP*
7	0	*NOP*

(c) *Step #9: Decode* IR = 210006 *into* MEM[6] ← A*, and set* PC *to* PC + 1 = 3.

CPU		
state	=	execute
PC	=	3
IR	=	210006
	=	MEM[6] ← A
A	=	30

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	30	*NOP*
7	0	*NOP*

(d) *Step #10: Execute the instruction* MEM[6] ← A*, i.e., set* MEM[6] *to* A = 30.

CPU		
state	=	fetch
PC	=	3
IR	=	100000
	=	
A	=	30

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	30	*NOP*
7	0	*NOP*

(e) *Step #11: Fetch the next instruction* IR = 100000 *from* PC = 3.

CPU		
state	=	decode
PC	=	4
IR	=	100000
	=	*HALT*
A	=	30

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	30	*NOP*
7	0	*NOP*

(f) *Step #12: Decode* IR = 100000 *into HALT, and set* PC *to* PC + 1 = 4.

Figure 7: *Computing the sum* 10 + 20*, as executed on a stored program computer.*

CPU		
state	=	execute
PC	=	4
IR	=	100000
	=	*HALT*
A	=	30

MEM		
Address	*Value*	*Meaning*
0	220004	A ← MEM[4]
1	300005	A ← A + MEM[5]
2	210006	MEM[6] ← A
3	100000	*HALT*
4	10	*NOP*
5	20	*NOP*
6	30	*NOP*
7	0	*NOP*

(a) *Step* #13*: Execute the instruction HALT, i.e., stop execution.*

Figure 8: *Computing the sum* 10 + 20*, as executed on a stored program computer.*

fetch the instruction from: it holds a number just like A does, but the number in PC will be used to keep track of where the next instruction is in memory. With this in mind, we also need to slightly alter the way programs are executed:

1. Encode a program P onto paper tape; load this tape into memory using the tape reader, then zero the program counter PC and start the computer.
2. From the address in PC, fetch the next instruction in the program and call it IR.
3. Increment PC so it points to the next instruction.
4. If
   (a) IR $=\perp$ (i.e., an invalid instruction is encountered) or
   (b) IR = `HALT` (i.e., the program concludes normally)

   then halt the computer, otherwise execute IR (i.e., perform the operation it specifies).
5. Repeat from line 2.

Notice that we have included a new step to update the value of PC (by adding one to it). This is analogous to ensuring we fetch the instruction from the next row of the tape in our previous design. Now reconsider the example program we looked at before, *and* how it is encoded:

$A \leftarrow$ MEM	$\mapsto$	220004
$A \leftarrow A +$ MEM	$\mapsto$	300005
MEM $\leftarrow A$	$\mapsto$	210006
HALT	$\mapsto$	100000

Execution using our new computer proceeds in more or less the same way, except that now it holds both data and instructions in MEM rather than having the instructions on a dedicated tape. As before, one can imagine drawing the state of the computer at each step in the execution; we do this in Figure 6 to Figure 8. Again, after step #13, the computer halts and we end up with the result of the addition, i.e., 30, stored in MEM.

Although reading and following executions of *existing* programs is a good start, writing your own programs is really the only way to get to grips with this topic.

See if you can write a program for the von Neumann computer that evaluates the expression

$$x \cdot (y + z)$$

where x, y and z are stored in memory at addresses of your choice; since the computer has no multiplication instruction, this demands some careful thought! Demonstrate that the program works by listing the steps (similar to above) for example x, y and z.

The current instruction set contains three rough classes of instruction: some perform load and store from and to memory, some perform arithmetic operations, and some perform updates to PC. Depending on what we use the computer for, other instructions might be useful of course: assuming you have a free choice, write the encoding *and* meaning for some other instructions you deem useful.

Why choose *these* instructions in particular? For instance, what might you use them for? There is a limit to how many new instructions we can add: can you explain one reason why?

2 Harvard versus von Neumann computers

With such a simple example program, it might seem as if the two styles of computer are more or less the same, i.e., neither has a massive advantage over the other. For example, they both compute 10 + 20 = 30 at the end of the day. But, and this is a big but, there are (at least) two subtle and key differences between them.

2.1 Beyond straight-line programs

With the Harvard-style design, we were forced to write instructions onto the tape and have them executed in the same order. There was, for example, no mechanism to "skip over" a given instruction or "jump" execution to an instruction based on a result we computed. Clearly this limits the sorts of program we could write. The good news is that the von Neumann-style upgrade removes this restriction by allowing *PC* to be altered by the program rather than just by the computer. By simply setting the value of *PC* we can direct execution to *any* instruction; we no longer have to write programs which are **straight-line**.

As a simple example, imagine we alter our example program by replacing the *HALT* instruction with one that alters the value of *PC*:

$$\begin{array}{lcl} A \leftarrow \mathsf{MEM} & \mapsto & 220004 \\ A \leftarrow A + \mathsf{MEM} & \mapsto & 300005 \\ \mathsf{MEM} \leftarrow A & \mapsto & 210006 \\ PC \leftarrow 0 & \mapsto & 400000 \end{array}$$

When we execute the last instruction, it sets the value of *PC* back to zero. Put more simply, it starts executing the program again right from the start. Writing down the steps of execution only, rather than drawing the state at each step, we now get:

Step #1: Load the tape into memory, set PC = 0 and start the computer.

Step #2: Fetch the next instruction IR = 220004 from PC = 0.

Step #3: Decode IR = 220004 into A ← MEM[4], and set PC to PC + 1 = 1.

Step #4: Execute the instruction A ← MEM[4], i.e., set A to MEM[4] = 10.

Step #5: Fetch the next instruction IR = 300005 from PC = 1.

Step #6: Decode IR = 300005 into A ← A + MEM[5], and set PC to PC + 1 = 2.

Step #7: Execute the instruction A ← A + MEM[5], i.e., set A to A + MEM[5] = 30.

Step #8: Fetch the next instruction IR = 210006 from PC = 2.

Step #9: Decode IR = 210006 into MEM[6] ← A, and set PC to PC + 1 = 3.

Step #10: Execute the instruction MEM[6] ← A, i.e., set MEM[6] to A = 30.

Step #11: Fetch the next instruction IR = 400000 from PC = 3.

Step #12: Decode IR = 400000 into PC ← 0, and set PC to PC + 1 = 4.

Step #13: Execute the instruction PC ← 0, i.e., set PC to 0.

Step #14: Fetch the next instruction IR = 220004 from PC = 0.

Step #15: Decode IR = 220004 into A ← MEM[4], and set PC to PC + 1 = 1.

Step #16: Execute the instruction A ← MEM[4], i.e., set A to MEM[4] = 10.

Step #17: ...

Clearly the program never finishes in the sense that we never execute a *HALT* instruction. What we have constructed is an **infinite loop**; in this situation the computer will seem to "freeze" since it ends up executing the same instructions over and over again forever [13].

Although the occurrence of an infinite loop is normally deemed a problem, can you think of an example where it might be *required*?

Even so, you might argue it would be useful to detect infinite loops so the computer can be manually halted. How might you approach doing so? For example, what changes to the computer might be necessary? Is your approach *always* guaranteed to work?

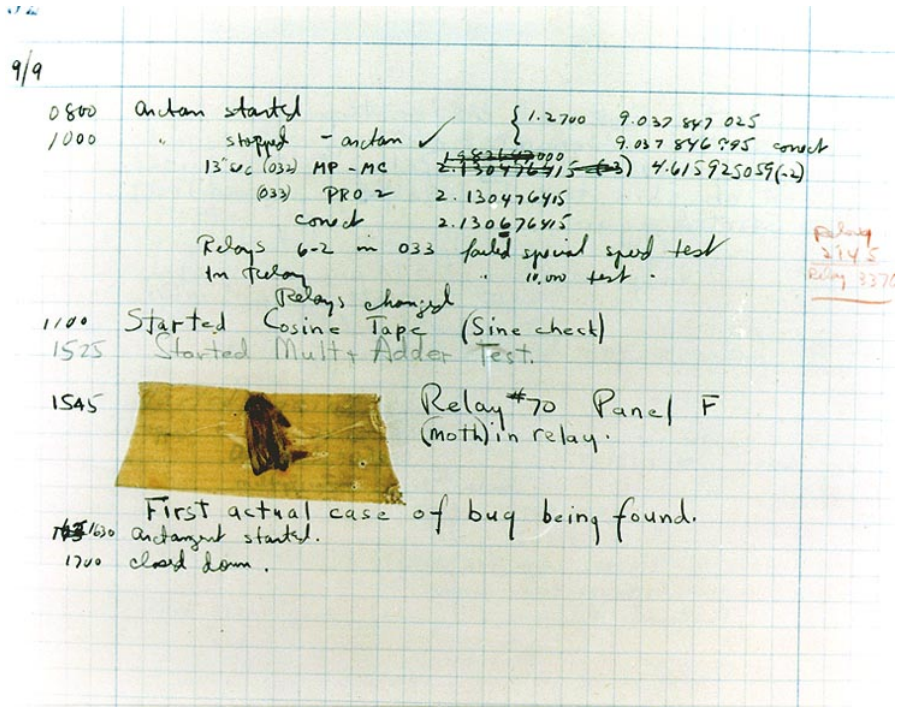

Figure 9: *A moth found by operations of the Harvard Mark 2; the "bug" was trapped within the computer and caused it to malfunction (public domain image, source:* http://en.wikipedia.org/wiki/File:H96566k.jpg*).*

2.2 Toward self-modifying programs

With the Harvard-style design, we were prevented from altering the tape content after we started execution. Put another way, programming happens strictly *before* program execution. However, with the von Neumann-style design instructions and data are the same thing so we can alter instructions *during* execution just as easily as data.

What does this mean in practise? Imagine we take the example program but introduce a **bug** [22]. A bug is a programming error: it is a mistake in a program that causes it to malfunction, behaving in a different way than we might have intended. The term bug (and **debug** [8], that relates to fixing the mistake) have well-discussed histories; a nice story relates them to a *real* bug (a moth) that famously short-circuited the Harvard Mark 2 computer! The bug here is that instead of storing the addition result in MEM, we mistakenly store it in MEM:

$$\begin{array}{lcl} A \leftarrow \mathsf{MEM} & \mapsto & 220004 \\ A \leftarrow A + \mathsf{MEM} & \mapsto & 300005 \\ \mathsf{MEM} \leftarrow A & \mapsto & 210003 \\ \mathit{HALT} & \mapsto & 100000 \end{array}$$

Again writing down the steps of execution only, we obtain:

Step #1: Load the tape into memory, set PC = 0 and start the computer.

Step #2: Fetch the next instruction IR = 220004 from PC = 0.

Step #3: Decode IR = 220004 into A ← MEM[4], and set PC to PC + 1 = 1.

Step #4: Execute the instruction A ← MEM[4], i.e., set A to MEM[4] = 10.

Step #5: Fetch the next instruction IR = 300005 from PC = 1.

Step #6: Decode IR = 300005 into A ← A + MEM[5], and set PC to PC + 1 = 2.

Step #7: Execute the instruction A ← A + MEM[5], i.e., set A to A + MEM[5] = 30.

Step #8: Fetch the next instruction IR = 210003 from PC = 2.

Step #9: Decode IR = 210003 into MEM[3] ← A, and set PC to PC + 1 = 3.

Step #10: Execute the instruction MEM[3] ← A, i.e., set MEM[3] to A = 30.

Step #11: Fetch the next instruction IR = 000030 from PC = 3.

Step #12: Decode IR = 000030 into *NOP*, and set PC to PC + 1 = 4.

Step #13: Execute the instruction *NOP*, i.e., do nothing.

Step #14: Fetch the next instruction IR = 000010 from PC = 4.

Step #15: Decode IR = 000010 into *NOP*, and set PC to PC + 1 = 5.

Step #16: Execute the instruction *NOP*, i.e., do nothing.

Step #17: ...

Step #11 is now different: something weird has happened because initially we had a *HALT* instruction in MEM, but when we come to fetch it, it has changed! The reason is obvious if we look a few steps back. In step #10 we execute the instruction which *should* store the result of our addition. But remember the bug: instead of storing the result in MEM, we store the result, i.e., 30, in MEM. Of course, MEM is part of the program, so as things progress we end up trying to *execute* the number 30 we previously stored. This behaviour is allowed because data and instructions are the same thing now. So we fetch the data value 30 and try to interpret it as an instruction according to our encoding.

To cut a long story short, the program has modified itself [21] by altering instructions relating to the same program as is being executed! In this case, the self-modification can only be a bad thing since without a *HALT* instruction the computer will never stop executing the program. More than likely it will "crash" somehow [6]. More generally however, the idea is that in other cases we can put self-modification to constructive use, i.e., have a program change instructions so that something useful happens as a result.

3 A self-modifying virus

Finally we can talk about our virus which, you will remember, aims to change itself to avoid detection by the virus scanner. Imagine our example computer includes a single instruction that, when executed, causes something bad to happen; maybe the computer self-destructs or something. For the sake of argument, we will say that whenever the computer executes the extra instruction 111111 this represents the payload of our virus. In other words, in our encoding 111111 means **payload**, and when **payload** is executed the virus has won.

It is not too hard to imagine that when we infect a file called F, the virus payload will then appear somewhere within it. If this is the case, the virus scanner can easily detect the virus by using the single element signature $S = \langle 111111 \rangle$ to scan F as we discussed at the beginning of the Chapter:

$$
\begin{array}{rcccccccccccccl}
F & = & \langle & \ldots & , & \ldots & , & \ldots & , & \ldots & , & 111111, & \ldots & \rangle & \\
S & = & \langle & & & 111111 & & & & & & & & \rangle & \mapsto \textbf{ no match} \\
 & & \langle & & & & & 111111 & & & & & & \rangle & \mapsto \textbf{ no match} \\
 & & \langle & & & & & & & 111111 & & & & \rangle & \mapsto \textbf{ no match} \\
 & & \langle & & & & & & & & & 111111 & & \rangle & \mapsto \textbf{ match}
\end{array}
$$

With this in mind, the virus writer has to somehow *hide* the virus payload so the virus scanner cannot detect it. To do this, we will use the properties of the XOR (short for "exclusive-or") function [9] we met in Chapter 2.

3.1 Using XOR to mask numbers

XOR can be used to mask (or hide) the value of a number. Imagine we have a bit x and we do not want anyone to know it; select a bit k and compute $y = x \oplus k$. If we give y to someone, can they tell us what x is? Well, if $y = 0$ then that could have been produced by us having $x = 0$ and selecting $k = 0$ *or* having $x = 1$ and selecting $k = 1$ because

$$y = 0 = x \oplus k = \begin{cases} 0 \oplus 0 & \text{if } x = 0 \text{ and } k = 0 \\ 1 \oplus 1 & \text{if } x = 1 \text{ and } k = 1 \end{cases}$$

Similarly, if $y = 1$ then that could have been produced by us having $x = 0$ and selecting $k = 1$ or having $x = 1$ and selecting $k = 0$ because

$$y = 1 = x \oplus k = \begin{cases} 0 \oplus 1 & \text{if } x = 0 \text{ and } k = 1 \\ 1 \oplus 0 & \text{if } x = 1 \text{ and } k = 0 \end{cases}$$

The point is, if we give someone y they can not tell us x for sure unless they also know k. Since we know k, we can easily tell what x is since $y \oplus k = x \oplus k \oplus k = x$. You can think of this as a limited form of encryption, except we call k the mask (rather than the key) and say x has been masked by k to produce y.

The problem is that XOR is a Boolean function: the inputs and output have to be either 0 or 1. To apply it to the decimal numbers in MEM, we first apply what we learnt in Chapter 2 to convert between decimal and binary. For example

$$\begin{aligned} 310000_{(10)} &= 01001011101011110000_{(2)} \\ 329975_{(10)} &= 01010000100011110111_{(2)} \end{aligned}$$

Now we can compute

$$\begin{aligned} 01001011101011110000_{(2)} \oplus 01010000100011110111_{(2)} &= 00011011001000000111_{(2)} \\ &= 111111_{(10)} \end{aligned}$$

simply by applying XOR to the corresponding bits. Likewise, since

$$\begin{aligned} 111111_{(10)} &= 00011011001000000111_{(2)} \\ 329975_{(10)} &= 01010000100011110111_{(2)} \end{aligned}$$

we can compute

$$\begin{aligned} 00011011001000000111_{(2)} \oplus 01010000100011110111_{(2)} &= 01001011101011110000_{(2)} \\ &= 310000_{(10)} \end{aligned}$$

More simply, in terms of our example above we start with $x = 310000$ and select $k = 329975$. Then, if we compute $x \oplus k = 310000 \oplus 329975$ we get $y = 111111$. We know k, so if we compute $y \oplus k = 111111 \oplus 329975$ then we get back $x = 310000$.

3.2 A virus that masks the payload

Now for the clever part: notice that

- 310000 corresponds to the encoded instruction 310000 which means $A \leftarrow A - \mathsf{MEM}$,
- 329975 corresponds to the encoded instruction 329975 which means $A \leftarrow A \oplus \mathsf{MEM}$, and
- 111111 corresponds to the encoded instruction 111111 which means **payload**.

So basically

$$111111 = 310000 \oplus 329975$$

means we can build the **payload** instruction by simply applying $\oplus$ to two innocent looking instructions. Our strategy for hiding the virus payload should now be obvious: we mask the payload instruction within F which we load into memory and start executing. While the program is executing, we unmask the payload instruction using self-modification, and then execute it. There is a fancy name for viruses like this: they are called polymorphic [18]. The first time a virus was seen using the technique was around 1990 when a virus called **1260** (referring to the length) was discovered.

Confused? Understanding why it works might be a puzzle, but either way the end result is the following program and the corresponding encoding:

$A \leftarrow \mathsf{MEM}$	$\mapsto$	220003
$A \leftarrow A \oplus \mathsf{MEM}$	$\mapsto$	320005
$\mathsf{MEM} \leftarrow A$	$\mapsto$	210003
$A \leftarrow A - \mathsf{MEM}$	$\mapsto$	310000
$HALT$	$\mapsto$	100000
$A \leftarrow A \oplus \mathsf{MEM}$	$\mapsto$	329975

Just by eye-balling the encoded program we can see that the virus payload does not appear, so if we use the virus scanner on it we fail to detect a virus:

F	=	⟨	...	,	220003,	320005,	210003,	310000,	100000,	329975,	...	⟩			
S	=	⟨			111111							⟩	↦	**no match**	
		⟨				111111						⟩	↦	**no match**	
		⟨					111111					⟩	↦	**no match**	
		⟨						111111				⟩	↦	**no match**	
		⟨							111111			⟩	↦	**no match**	
		⟨								111111		⟩	↦	**no match**	

But so what? If the program does not contain the virus payload, nothing bad can happen right?! Unfortunately not. Have a look at what happens when we execute the program:

Step #1: Load the tape into memory, set PC = 0 and start the computer.

Step #2: Fetch the next instruction IR = 220003 from PC = 0.

Step #3: Decode IR = 220003 into A ← MEM[3], and set PC to PC + 1 = 1.

Step #4: Execute the instruction A ← MEM[3], i.e., set A to MEM[3] = 310000.

Step #5: Fetch the next instruction IR = 320005 from PC = 1.

Step #6: Decode IR = 320005 into A ← A ⊕ MEM[5], and set PC to PC + 1 = 2.

Step #7: Execute the instruction A ← A ⊕ MEM[5], i.e., set A to A ⊕ MEM[5] = 111111.

Step #8: Fetch the next instruction IR = 210003 from PC = 2.

Step #9: Decode IR = 210003 into MEM[3] ← A, and set PC to PC + 1 = 3.

Step #10: Execute the instruction MEM[3] ← A, i.e., set MEM[3] to A = 111111.

Step #11: Fetch the next instruction IR = 111111 from PC = 3.

Step #12: Decode IR = 111111 into **payload**, and set PC to PC + 1 = 4.

Step #13: Execute the instruction **payload**, i.e., do something bad.

Step #14: ...

Step #13 executes **payload** somehow, so the virus payload has been triggered even though it did not appear in F. Look at what happens step-by-step:

- Step #4 loads the masked virus payload, which looks like a $A \leftarrow A -$ MEM instruction, from MEM and into A.
- Step #7 unmasks it using an XOR instruction that applies the mask value k stored in MEM just like we saw above.
- At this point we have $A = 111111$ which is then stored back into MEM by step #10.
- Step #13 executes the unmasked instruction from MEM, at which point the payload is triggered and the virus wins.

The current instruction set includes instructions to load from and store into *fixed* addresses. For instance MEM ← A stores A at the address n which is fixed when we write the program. For computers such as ENIAC, self-modifying programs were often used in a positive way to extend this ability to *variable* addresses; put simply, this allows access to MEM for some i chosen during execution.

Imagine a sequence of numbers $X = \langle X_0, X_1, \ldots, X_{m-1} \rangle$ is stored at a known address in memory, and we want to compute the sum

$$\sum_{i=0}^{i<m} X_i.$$

If the sequence has 10 elements and starts at address 100, say, one way to do this would be

$$\begin{aligned} A &\leftarrow \quad \text{MEM} \\ A &\leftarrow A + \text{MEM} \\ &\vdots \\ A &\leftarrow A + \text{MEM} \end{aligned}$$

where clearly 100 + 1 = 101 is fixed, so we can write an appropriate instruction.

But what if m is *really* large, or only known once the program starts to execute? In these cases, we might prefer to write a loop: based on an m also stored in memory somewhere (at address 99 say), use the concept of self-modifying programs to compute the same result by looping through each i (like the loop in Chapter 3) and hence accumulating each X_i.

3.3 Preventing the virus without a virus scanner

The obvious question is, if the virus scanner we have does not work in this case then what *can* we do to stop the virus? In reality, this is quite an open question. On real computers we still do not have a definitive, general solution against viruses and malware. However, in the particular case of *our* virus we can think of a few possibilities:

- We could monitor the program during execution and prevent the virus payload being executed. So, for example, execution of *every* instruction would have to include an extra step saying "if *IR* is the virus payload then stop execution". But of course in reality there is not just one instruction that does something bad: more usually a subtle combination of many instructions will represent the payload, so this approach would not work. It also has the drawback of reducing the speed at which we can execute *all* programs, which might be quite unpopular.
- We could try to be more clever at coming up with the signature that identifies this sort of virus. We cannot use the masked payload 310000 because that is a valid instruction which basically *any* program might include! We cannot use the mask 329975 because this could be anything: the virus is free to choose any mask it wants (although obviously this alters the masked payload). One option could be to try and identify the pattern of self-modifying code. For example, if the virus scanner knew the meaning of instructions it could perhaps analyse the program and work out what is really going on. Clearly this means the virus scanner needs to be much more sophisticated and depends on the fact that writing similar code in a different way is not possible.
- We could alter the computer so that writing self-modifying programs is impossible. Modern computers include a scheme which roughly works by adding an extra "tag" to each element of MEM. If the tag is 0, this indicates that the element can be written to but *not* executed as an instruction; if the flag is 1, this means the element can be executed as an instruction but *not* written to. In terms of our example virus, this would mean the memory looks like:

MEM			
Address	*Tag*	*Value*	*Meaning*
0	1	220003	$A \leftarrow \mathsf{MEM}$
1	1	320005	$A \leftarrow A \oplus \mathsf{MEM}$
2	1	210003	$\mathsf{MEM} \leftarrow A$
3	1	310000	$A \leftarrow A - \mathsf{MEM}$
4	1	100000	*HALT*
5	0	329975	$A \leftarrow A \oplus \mathsf{MEM}$

So basically, we could not execute the virus because MEM could not be written to once the program was loaded. Put another way, the instruction $\mathsf{MEM} \leftarrow A$ would cause the computer to halt due to a violation of the rules. The drawback is that there *are* reasonable uses for self-modifying programs; by implementing this solution we prevent those from being executed as well. Plus, we have to make our memory, and the mechanism that accesses it, more complicated and therefore more costly.

References

[1] *Wikipedia: Alan Turing*. http://en.wikipedia.org/wiki/Alan_Turing (see p. 4).

[2] *Wikipedia: Anti-virus software*. http://en.wikipedia.org/wiki/Antivirus_software (see p. 4).

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

[4] *Wikipedia: Central Processing Unit (CPU)*. http://en.wikipedia.org/wiki/Central_processing_unit (see p. 5).

[5] *Wikipedia: Computer virus*. http://en.wikipedia.org/wiki/Computer_virus (see p. 3).

[6] *Wikipedia: Crash*. http://en.wikipedia.org/wiki/Crash_(computing) (see p. 16).

[7] *Wikipedia: Creeper*. http://en.wikipedia.org/wiki/Creeper_(program) (see p. 3).

[8] *Wikipedia: Debugging*. http://en.wikipedia.org/wiki/Debugging (see p. 15).

[9] *Wikipedia: Exclusive OR*. http://en.wikipedia.org/wiki/XOR (see p. 16).

[10] *Wikipedia: Halting problem*. http://en.wikipedia.org/wiki/Halting_problem (see p. 4).

[11] *Wikipedia: Harvard architecture*. http://en.wikipedia.org/wiki/Harvard_architecture (see p. 8).

[12] *Wikipedia: Harvard Mark I*. http://en.wikipedia.org/wiki/Harvard_Mark_I (see p. 8).

[13] *Wikipedia: Infinite loop*. http://en.wikipedia.org/wiki/Infinite_loop (see p. 14).

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

[15] *Wikipedia: Keystroke logging*. http://en.wikipedia.org/wiki/Keystroke_logging (see p. 3).

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

[17] *Wikipedia: Morris worm*. http://en.wikipedia.org/wiki/Morris_worm (see p. 3).

[18] *Wikipedia: Polymorphic code*. http://en.wikipedia.org/wiki/Polymorphic_code (see p. 17).

[19] *Wikipedia: Programmed Data Processor (PDP)*. http://en.wikipedia.org/wiki/Programmed_Data_Processor (see p. 3).

[20] *Wikipedia: Punched tape*. http://en.wikipedia.org/wiki/Punched_tape (see p. 6).

[21] *Wikipedia: Self-modifying code*. http://en.wikipedia.org/wiki/Self-modifying_code (see p. 16).

[22] *Wikipedia: Software bug*. http://en.wikipedia.org/wiki/Software_bug (see p. 15).

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

[24] *Wikipedia: Timeline of computer viruses and worms*. http://en.wikipedia.org/wiki/Timeline_of_computer_viruses_and_worms (see p. 3).

[25] *Wikipedia: Turing machine*. http://en.wikipedia.org/wiki/Turing_machine (see p. 4).

[26] *Wikipedia: von Neumann architecture*. http://en.wikipedia.org/wiki/Von_Neumann_architecture (see p. 9).

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.