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

“MY BUFFER RUNNETH OVER”

Here is a simple (or at least simple *looking*) question: if we take two sequences

$$T = \langle 0, 1 \rangle$$

and

$$S = \langle 2, 3, 4 \rangle,$$

what happens if we copy all the elements from S into T? At face value, this seems like a reasonable thing to do: afterwards we expect the sequences to be equal, i.e., that $T_0 = S_0 = 2$, $T_1 = S_1 = 3$ and $T_2 = S_2 = 4$. However, a problem is lurking. Namely, S_0, S_1 and S_2 are valid elements since $|S| = 3$, but $|T| = 2$ so T_2 is not valid. In other words, we would put *too much* content [10] into T. When defining how sequences can be accessed we include an implicit **bounds check** [2] so before each step, we implicitly do something like “if $0 \leq i < |T|$ and $0 \leq i < |S|$ then copy S_i into T_i, otherwise cause an error” behind the scenes. So to cut a long story short, an error occurs because trying to assign T_2 the value 4 is problematic.

Next question: does this match what a computer does when executing a program to copy S into T? The answer is no, not really. If you look at the instruction set from Chapter 4, specifically at the few instructions which access memory, the bounds check is missing; the computer just accesses memory without checking the address we give it. Suppose T and S are stored in memory somewhere, e.g., something like

$$\begin{array}{lcccccccccc} i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & \ldots & \\ \mathsf{MEM} & = & \langle & \ldots, & 0, & 1, & 2, & 3, & 4, & \ldots & \rangle \\ & = & & & T_0 & T_1 & S_0 & S_1 & S_2 & & \end{array}$$

If we write a program that tries to store a value in MEM, i.e., what it *thinks* should be T_2, something *else* is overwritten: in this case S_0. Clearly this is not ideal; the program has a **bug** [14] or mistake in it which will probably causes it to malfunction. That is, it will overwrite some data or an instruction that will be wrong if subsequently loaded. In this case, if we load S_0 after the copying has finished we end up with 4 rather than 2 because it was overwritten.

One might argue that this is a simple problem to solve: just avoid writing programs with bugs in! This is more tricky than you might think. Large programs are complex and human programmers *will* make mistakes, so bugs that cause *functional* problems in a program, i.e., mean it does not do what it should, are irritatingly common. However, the issue is magnified further if the bug implies a *security* problem, which of course is our focus. Specifically, what if an attacker intentionally tries to find and trigger bugs in our program so they can cause something “bad” to happen, e.g., the program allows access to a file which could not access otherwise? Given the overwriting bug might occur in a program, the real question is *can* it produce a security problem, and if so *how*?

Of course the answer to the first part is yes; the rest of this Chapter tries to explain the answer to the second part. Within the context of security, the overwriting bug is more generically called a **buffer overflow** [3], the idea being we overflow (i.e., put too much content into) a buffer (i.e., the memory allocated for something). The issue of buffer overflows has been described as the “vulnerability of the decade” since they act as an underlying technique that supports all sorts of other security issues. For example, the **Morris Worm** [9] mentioned in Chapter 4 used a buffer overflow in the `fingerd` server on UNIX-based computers, among other techniques, to propagate itself. Clearly there are ways to prevent or limit such attacks, but they have remained

```
algorithm MAIN begin
    $a \leftarrow 10$
    $b \leftarrow 20$
    $c \leftarrow$ ADD$(a, b)$
    return
end
```

(a) *The* MAIN *algorithm.*

```
algorithm ADD$(x, y)$ begin
    $z \leftarrow x + y$
    return $z$
end
```

(b) *The* ADD *algorithm.*

Figure 1: *Two algorithms used as a running example.*

a clear and present danger for more than twenty years; the cost resulting from the Morris Worm was estimated at up to $100 million for example.

There are a huge range of attacks and techniques that fall under this more general umbrella, but the basic idea is not hard to describe. Suppose we are an attacker, and we want to attack a web-server. A web-server could be **privileged** [12] in the sense that it can do things a normal program cannot; for example, it might have access to files that a normal program does not. Normally, we would interact with the web-server via a web-browser. Behind the scenes, the web-server and web-browser interact by simply sending commands and responses between each other in the form of strings. Something like "please send me web-page X", just less polite. Both programs use a data structure to store these strings: if they are written in C, they would use the C-string method from Chapter 5. The basic idea of a buffer overflow attack is for us, i.e., the attacker, to send a large string S to the web-server: the hope is that it copies the string into some buffer T which is too small to hold the content, thereby overwriting something else. The tricky part is to make the bug do something useful for us rather than simply crash the web-server. Typically, the idea is that we, i.e., the attacker, fix things up so that the bug causes some *other* instructions to be executed rather than those representing the web-server program. That is, the bug causes the program to jump somewhere unexpected. Ideally, we end up executing instructions of our choosing so that in short, we force execution of *our* program rather than the web-server. Of course this is difficult, but if we get everything just right we can execute a program of our choice using the privileges of the web-server. This might mean we can access a web-page we could not access otherwise: all manner of bad things are possible.

1 From algorithms to sub-routines

Imagine we write two simple algorithms, one of which invokes the other; the algorithms are shown in Figure 1. We can easily work out what should happen if we invoke MAIN:

Step #1 Assign $a \leftarrow 10$.

Step #2 Assign $b \leftarrow 20$.

Step #3 Invoke ADD(a, b), i.e., invoke ADD$(10, 20)$.

  Step #3.1 Assign $z \leftarrow x + y$, i.e., assign $z \leftarrow 10 + 20 = 30$.

  Step #3.2 Return z, i.e., return 30.

  then assign $c \leftarrow 30$.

Step #4 Return.

Basically, MAIN assigns values to a and b, and then passes them as input to an invocation of ADD. The arguments are added together and their sum z is produced as output returned to MAIN; this is assigned to c. The point is, there are some crucial details missing if we think about similar steps within a corresponding program: exactly *how* are a and b are passed from MAIN to ADD, and *how* is z returned back to MAIN?

1.1 A simple approach to sub-routine calls

1.1.1 Attempt #1: a simple starting point

Imagine we implement MAIN and ADD in the program described by Figure 2. There are two parts:

1. MAIN is represented by the instructions held in addresses #0 . . . #11, and uses a, b and c held in addresses #12 . . . #24.

Address	Instruction			Purpose
0	200010	$\mapsto$	A ← 10	compute $a = 10$
1	210012	$\mapsto$	MEM[12] ← A	store a
2	200020	$\mapsto$	A ← 20	compute $b = 20$
3	210013	$\mapsto$	MEM[13] ← A	store b
4	220012	$\mapsto$	A ← MEM[12]	load a
5	210019	$\mapsto$	MEM[19] ← A	store $x = a$
6	220013	$\mapsto$	A ← MEM[13]	load b
7	210020	$\mapsto$	MEM[20] ← A	store $y = b$
8	400015	$\mapsto$	PC ← 15	call
9	220021	$\mapsto$	A ← MEM[21]	load z
10	210014	$\mapsto$	MEM[14] ← A	store $c = z$
11	100000	$\mapsto$	*HALT*	halt
12	000000	$\mapsto$	*NOP*	a
13	000000	$\mapsto$	*NOP*	b
14	000000	$\mapsto$	*NOP*	c
15	220019	$\mapsto$	A ← MEM[19]	load x
16	300020	$\mapsto$	A ← A + MEM[20]	compute $z = x + y$
17	210021	$\mapsto$	MEM[21] ← A	store z
18	400009	$\mapsto$	PC ← 9	return
19	000000	$\mapsto$	*NOP*	x
20	000000	$\mapsto$	*NOP*	y
21	000000	$\mapsto$	*NOP*	z

Figure 2: *Attempt* #1 *at implementing* Main *and* Add.

2. Add is represented by the instructions held in addresses #15 . . . #18, and uses x, y and z held in addresses #19 . . . #21.

Each of the parts would be termed a **sub-routine** [7]. Well, actually we would more usually term them **functions** but this name makes them sound like Mathematical functions: in Chapter 3 we already described why there might be important differences between the two. So for the sake of avoiding confusion, we stick with sub-routine as a name because either way the idea is the same: the sub-routines capture steps required to complete a specific task somewhat independently from the rest of the program. While algorithms are invoked, sub-routines are **called**: the implementation of Main would be the **caller** (i.e., the part that does the calling), Add the **callee** (i.e., the part that is called).

The program itself is a lot longer than those we have looked at before, but if you read down the right-hand column, the instructions loosely replicate lines of the original algorithms. A step-by-step description of execution is too verbose to include, but here is an abridged version including just the most important aspects:

1. The instructions at addresses #0 . . . #3 act to assign a and b the initial values of 10 and 20.

2. The instructions at addresses #4 . . . #7 copy a and b into x and y, ready for use as arguments by Add; control is then passed to Add when the instruction at address #8 sets $PC = 15$.

3. The instructions at addresses #15 . . . #17 compute $z = x + y = 30$; the instruction at address #18 then passes control back to Main by setting $PC = 9$.

4. The instructions at addresses #9 . . . #10 copy the return value z into c and the program is then halted by the instruction at address #11; the result is that $c = 30$.

1.1.2 Problem #1: multiple callers

Algorithms can be invoked from more than one place. In a sense, this is also one of the main attractions of using sub-routines in a program: they allow us to *describe* how to perform some task once, and then *use* that description as many times as we like. Clearly this is more attractive than writing out the same lines every time we need to use them.

Imagine we take our original Main algorithm and alter it slightly to produce Figure 3; the crucial difference is that it now invokes Add *twice* to compute $a + b$ and then $d + e$. This highlights a problem: if we follow the same implementation strategy as with the original Main and Add, we will get confused when returning from Add. How do we know where to return *to*? In our previous implementation, we fixed the address of where the next instruction in Main should be after we finish Add; now there are *two* "next instructions". Clearly this

```
algorithm MAIN begin
    $a \leftarrow 10$
    $b \leftarrow 20$
    $c \leftarrow$ ADD$(a, b)$
    $d \leftarrow 30$
    $e \leftarrow 40$
    $f \leftarrow$ ADD$(d, e)$
    return
end
```

Figure 3: *An altered* MAIN *that now invokes* ADD *twice rather than once.*

```
algorithm FIBONACCI-ITERATIVE($n$) begin
    $i \leftarrow 1$
    $p \leftarrow 0$
    $q \leftarrow 1$
    while $i < n$ do
        $i \leftarrow i + 1$
        $t \leftarrow p + q$
        $p \leftarrow q$
        $q \leftarrow t$
    end
    return $q$
end
```

(a) *An iterative approach.*

```
algorithm FIBONACCI-RECURSIVE($n$) begin
    if $n = 0$ then
        return 0
    end
    if $n = 1$ then
        return 1
    end
    $p \leftarrow$ FIBONACCI-RECURSIVE$(n - 1)$
    $q \leftarrow$ FIBONACCI-RECURSIVE$(n - 2)$
    return $p + q$
end
```

(b) *A recursive approach.*

Figure 4: *Two algorithms to compute the n-th Fibonacci number.*

is a problem that needs to be solved, preferably without having to write two different implementations of ADD since that defeats the purpose of reusing one description.

1.1.3 Problem #2: local variables

The concept of **recursion** [13] is commonplace in nature, and a great analogy is provided by the **Droste effect** [5]: basically this is where an image contains a smaller instance of itself, with the name taken from a Dutch brand of hot chocolate mixture whose packaging is shown in Figure 5.

By replacing "image" with "algorithm" the concept still makes sense in that a recursive algorithm is one that contains invocations of itself. At a high level, the idea is to divide a problem into smaller sub-problems of the same type, then solve the sub-problems: the smaller solutions can then be combined to solve the original problem. Since the sub-problems will be solved using the same algorithm as the original, recursive algorithms tend to be very concise and admired for their elegance. However, despite their advantages, the concept can be hard to grasp: an example will make this a lot easier. Consider computation of the **Fibonacci numbers** [6] which form a sequence defined by

$$\begin{array}{rcl} F_0 & = & 0 \\ F_1 & = & 1 \\ F_i & = & F_{i-1} + F_{i-2} \end{array}$$

That is, the first two numbers in the sequence are fixed and the rest we compute out: each i-th element in the sequence is just the sum of the previous two. So for example

$$\begin{array}{rcccccc} F_2 & = & F_1 + F_0 & = & 1 + 0 & = & 1 \\ F_3 & = & F_2 + F_1 & = & 1 + 1 & = & 2 \\ F_4 & = & F_3 + F_2 & = & 2 + 1 & = & 3 \\ & \vdots & & \vdots & & \vdots & \end{array}$$

Put another way, to compute F_i we invoke the same algorithm to solve the smaller sub-computations F_{i-1} and F_{i-2} then combine the solutions by adding them together. That should sound sensible, but what does an algorithm to compute the n-th Fibonacci number look like?

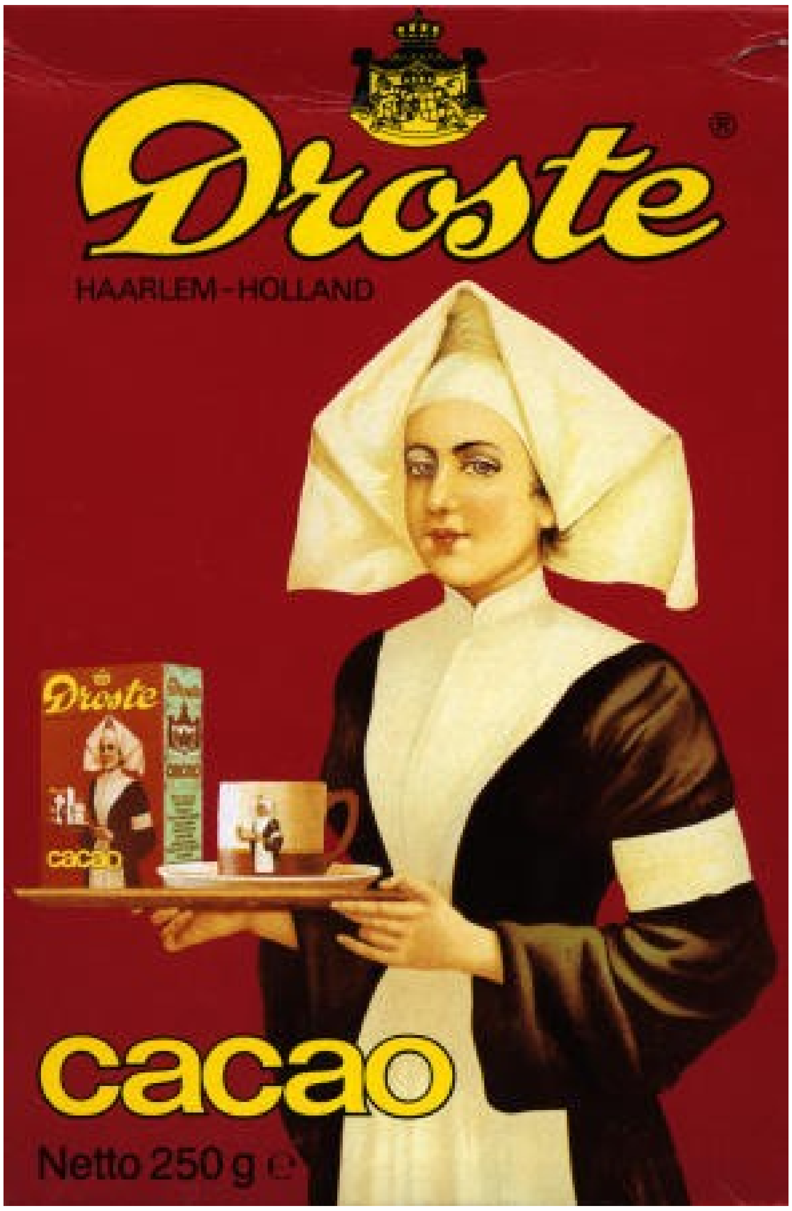

Figure 5: *The Droste effect whereby an image contains a smaller instance of itself (public domain image, source: http://en.wikipedia.org/wiki/File:Droste.jpg)*

1. We could adopt an **iterative** approach: this basically means that we write a loop that computes successive Fibonacci numbers until it finds the one we want. The algorithm Fibonacci-Iterative in Figure 4a uses this approach; imagine we invoke it with $n = 3$:

   Step #1 Assign $i \leftarrow 1$, $p \leftarrow 0$, $q \leftarrow 1$.

   Step #2 Since $i < n$, perform the next loop iteration.

   Step #3 Assign $i \leftarrow i + 1 = 2$, $t \leftarrow p + q = 0 + 1 = 1$, $p \leftarrow q = 1$, $q \leftarrow t = 1$.

   Step #4 Since $i < n$, perform the next loop iteration.

   Step #5 Assign $i \leftarrow i + 1 = 3$, $t \leftarrow p + q = 1 + 1 = 2$, $p \leftarrow q = 1$, $q \leftarrow t = 2$.

   Step #6 Since $i = n$, stop the loop.

   Step #7 Return q, i.e., return 2.

2. We could adopt an **recursive** approach: this basically means that we write an algorithm that invokes itself to solve small sub-problems (a **recursive case**) until it finds one where the solution can be returned trivially (a **base case**). Fibonacci-Recursive in Figure 4b uses this approach; imagine we invoke it with $n = 3$:

   Step #1 Since $n \neq 0$ and $n \neq 1$, continue.

   Step #2 Invoke Fibonacci-Recursive($n - 1$), i.e., invoke Fibonacci-Recursive(2).

     Step #2.1 Since $n \neq 0$ and $n \neq 1$, continue.

     Step #2.2 Invoke Fibonacci-Recursive($n - 1$), i.e., invoke Fibonacci-Recursive(1).

       Step #2.2.1 Since $n = 1$, return 1.

       and assign $p \leftarrow 1$

     Step #2.3 Invoke Fibonacci-Recursive($n - 2$), i.e., invoke Fibonacci-Recursive(0).

       Step #2.3.1 Since $n = 0$, return 0.

       and assign $q \leftarrow 0$

     Step #2.4 Return $p + q$, i.e., return $1 + 0 = 1$.

     and assign $p \leftarrow 1$.

   Step #3 Invoke Fibonacci-Recursive($n - 2$), i.e., invoke Fibonacci-Recursive(1).

     Step #3.1 Since $n = 1$, return 1.

     and assign $q \leftarrow 1$.

   Step #4 Return $p + q$, i.e., return $1 + 1 = 2$.

Implement (task #1)

It makes sense to explore the concept of recursion by writing your own recursive algorithms. We have already seen two iterative candidates that could be translated into a recursive alternative:

1. an algorithm for finding the length of a string, e.g., C-String-Length from Chapter 5, or
2. an algorithm for performing exponentiation, e.g., Exponentiate-Horner from Chapter 3.

Try to write the recursive alternatives, and show how they work using the same example inputs as originally.

Since both approaches compute the same result, selecting between them could be viewed as a matter of taste. Ignoring this however, imagine we write a program that uses recursion; in particular, imagine the program corresponds to Fibonacci-Recursive. In the algorithm we can only perform one step at a time; the same is true for the program. But, in a sense, more than one instance of Fibonacci-Recursive is *active* at a given step: looking at the example, step #2.2.1 is a step in an invocation of Fibonacci-Recursive nested inside another one, and another one again! Each active instance has some state we need to keep track of. We need to remember that the value of p assigned to by step #2.2 does not conflict with that resulting from step #2: each p is "owned" by the associated instance of Fibonacci-Recursive.

This discussion of how the algorithms behave highlights a problem: if we follow the same implementation strategy as we did previously with Main and Add, we will confuse the values of p. In Figure 2 we only have *one* place for z; the analogy of z here is p, so basically one instance of the sub-routine would overwrite the p owned by another. Clearly this is another problem that needs a solution.

2 Constructing an improved sub-routine call mechanism

2.1 A stack data structure

A **stack** [15] is an important data structure in Computer Science. The easiest way to think about a stack is as a "container" that organises data in a certain way: we can add items to the stack and remove items from it, and the stack basically keeps track of where all the items are. Sometimes a stack is called a **First-In Last-Out (FILO)**, but another way to say the same thing would be to call it a **Last-In First-Out (LIFO)**: the idea is that we can "push" (or add) items onto the stack, but when we "pop" (or remove) items off the stack we get the data on the **Top of Stack (ToS)**, i.e., the last item pushed. To get the required behaviour we need three components, namely

1. an address, called the **Stack Pointer (SP)**, that keeps track of where the ToS is,
2. an algorithm called Push that takes some item x and adds it to the stack, and
3. an algorithm called Pop that removes an item and returns it as a result.

Based on the fact that SP is just a number we keep somewhere, the two algorithms we need are actually quite simple:

```
algorithm Push($x$) begin
    MEM $\leftarrow x$
    $SP \leftarrow SP - 1$
    return
end
```

```
algorithm Pop begin
    $SP \leftarrow SP + 1$
    return MEM
end
```

To reinforce what is going on, consider a simple example where we have the following initial memory content and set $SP = 7$

$$\begin{array}{rcllllllll}
SP & & & & & & & \downarrow & & \\
i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & \ldots \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & \ldots & \rangle
\end{array}$$

Notice that we have marked where the current value of SP says the ToS is to make things easier to read, and that stacks traditionally grow downwards as items are added (i.e., from higher address to lower addresses) so we follow this. By invoking Push(104), our intention is to add 104 to the stack; the behaviour of the algorithm is simple

Step #1 Assign MEM $\leftarrow$ 104.

Step #2 Assign $SP \leftarrow SP - 1 = 6$.

After it terminates, we end up with $SP = 6$ and the memory content

$$\begin{array}{rcllllllll}
SP & & & & & & & \downarrow & & \\
i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & \ldots \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 104, & \ldots & \rangle
\end{array}$$

We can carry on pushing more items if we want; for example, after two more invocations Push(101) and Push(108) we end up with $SP = 4$ and the memory content

$$\begin{array}{rcllllllll}
SP & & & & & \downarrow & & & & \\
i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & \ldots \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 108, & 101, & 104, & \ldots & \rangle
\end{array}$$

What happens if we now try to remove an item? If we invoke Pop then our intention is to remove (i.e., retrieve) the last item we pushed. The behaviour we get is again simple

Step #1 Assign $SP \leftarrow SP + 1 = 5$.

Step #2 Return MEM, i.e., return 108.

and we get what we expected: 108 was the last item previously pushed. After the algorithm terminates we end up with $SP = 5$ and the memory content

$$\begin{array}{rcccccccccc}
SP & & & & & & \downarrow & & & & \\
i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 108, & 101, & 104, & \ldots & \rangle
\end{array}$$

Notice that we have not deleted 108: it is still there in memory. All that has happened is that by updating SP, the organisation now excludes whatever is in MEM from being part of the stack data structure.

Going back to the topic of sub-routines, our approach to solving the problems we identified will be to use a stack just like this one. To illustrate the concept, imagine trying to pass an argument from Main to Add: we just make Main push the argument onto the stack, and then make Add pop it off again. Employing a more diagrammatic description, invoking Main would produce something like the following behaviour:

	Main		Add
Step #1	Assign $a \leftarrow 10$		
Step #2	Assign $b \leftarrow 20$		
Step #3	Invoke Push(b)		
Step #4	Invoke Push(a)		
Step #5	Pass control to Add	$\rightarrow$	
Step #6			Assign $x \leftarrow$ Pop()
Step #7			Assign $y \leftarrow$ Pop()
Step #8			Invoke Push($x + y$)
Step #9		$\leftarrow$	Pass control to Main
Step #10	Assign $c \leftarrow$ Pop()		

Notice that because of the way the stack works, we push b *then* a (i.e., sort of in reverse) so that when we pop x then y in Main, they get assigned the right values. To prove this really works, we need to look at the gruesome detail of how the algorithms progress. Imagine we reset $SP = 7$ and the memory content to

$$\begin{array}{rcccccccccc}
SP & & & & & & & & \downarrow & & \\
i & = & & \ldots, & 3, & 4, & 5, & 6, & 7, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & \ldots & \rangle
\end{array}$$

If we invoke Main then the resulting behaviour modelled by the diagram above is given by the following:

Step #1 Assign $a \leftarrow 10$.

Step #2 Assign $b \leftarrow 20$.

Step #3 Invoke Push(b), i.e., invoke Push(20).

> **Step #3.1** Assign $\mathsf{MEM}_{SP} \leftarrow 20$.
>
> **Step #3.2** Assign $SP \leftarrow SP - 1 = 6$.

Step #4 Invoke Push(a), i.e., invoke Push(10).

> **Step #4.1** Assign $\mathsf{MEM}_{SP} \leftarrow 10$.
>
> **Step #4.2** Assign $SP \leftarrow SP - 1 = 5$.

Step #5 Pass control to Add.

> **Step #5.1** Invoke Pop().
>
> > **Step #5.1.1** Assign $SP \leftarrow SP + 1 = 6$.
> >
> > **Step #5.1.2** Return MEM_{SP}, i.e., return 10.
> >
> > then assign $x \leftarrow 10$.
>
> **Step #5.2** Invoke Pop().
>
> > **Step #5.2.1** Assign $SP \leftarrow SP + 1 = 7$.
> >
> > **Step #5.2.2** Return MEM_{SP}, i.e., return 20.
> >
> > then assign $y \leftarrow 20$.
>
> **Step #5.3** Assign $z \leftarrow x + y$, i.e., assign $z \leftarrow 10 + 20 = 30$.

Step #5.4 Invoke PUSH(z), i.e., invoke PUSH(30).

Step #5.4.1 Assign MEM $\leftarrow$ 30.

Step #5.4.2 Assign $SP \leftarrow SP - 1 = 6$.

Step #5.5 Pass control to MAIN.

Step #6 Invoke POP().

Step #6.1 Assign $SP \leftarrow SP + 1 = 7$.

Step #6.2 Return MEM, i.e., return 30.

then assign $c \leftarrow 30$.

On one hand this description is quite long and therefore not so nice to read through, but on the other hand there is no longer any missing detail: the advantage we now have is that provided the stack exists, there is no hidden mystery behind the process of MAIN invoking ADD.

2.2 Adding a stack to the example computer

To support the approach described above, we need a minor upgrade to the computer. None of the changes are particularly radical, with the main goal being addition of instructions to manage and access stack content:

1. We add some new instructions that allow us to move PC into A and vice versa:

   - 43$nnnn$ means PC $\leftarrow$ A.
   - 44$nnnn$ means A $\leftarrow$ PC.

   Before this upgrade, we could only set PC to fixed addresses via an instruction of the form

   $$PC \leftarrow n.$$

   Now we can compute an address in A during execution of the program, and set PC to this new value.

2. We introduce another accumulator called SP which represents the stack pointer; when the computer is reset, we assume SP is set to the highest address in memory (in the same way PC is set to 0, the lowest address).

3. We introduce two new instructions:

   - 50$nnnn$ means MEM[SP] $\leftarrow$ A, SP $\leftarrow$ SP $-$ 1.
   - 51$nnnn$ means SP $\leftarrow$ SP $+$ 1, A $\leftarrow$ MEM[SP].

   which essentially to the same thing as the PUSH and POP algorithms: they put the current value of A onto the ToS and set the value of A to that from the ToS, updating SP accordingly.

4. We add some new instructions that allow us to use SP in a wider variety of ways, mainly in line with how we already use A:

   - 52$nnnn$ means MEM[SP $+ n$] $\leftarrow$ A.
   - 53$nnnn$ means A $\leftarrow$ MEM[SP $+ n$].
   - 54$nnnn$ means A $\leftarrow$ A $+$ MEM[SP $+ n$].
   - 55$nnnn$ means A $\leftarrow$ A $-$ MEM[SP $+ n$].
   - 56$nnnn$ means A $\leftarrow$ A $\oplus$ MEM[SP $+ n$].
   - 57$nnnn$ means SP $\leftarrow$ SP $+ n$.
   - 58$nnnn$ means SP $\leftarrow$ SP $- n$.

   For example, now we can perform loads and stores relative to the current value of SP, i.e., directly access the stack content. You can think of this as bending the rules a little: with these new instructions, we are not strictly limited to accessing the ToS only.

2.3 A stack-based approach to sub-routine calls

Armed with a more capable computer, improving on our original strategy for sub-routine calls is fairly straightforward. The basic idea is that each time we call a sub-routine, we create some space for it on the stack called a **stack frame** [4] (or sometimes an **activation record**); each time a sub-routine returns, we remove the associated stack frame. The stack frame itself is home to various items of data, for example

1. a **Return Address (RA)**, i.e., where the sub-routine should return to once it finishes,
2. any incoming **arguments** and outgoing **Return Value (RV)**, and
3. any **local** variables, i.e., variables it "owns".

We takes things step-by-step, starting with the original program (i.e., attempt #1) and making a series of small changes to end up with a better program; each step essentially adds one of the items above to the stack frame, each addition makes things a little more complicated but eventually solve the problems we started off with.

2.3.1 Attempt #2: using return addresses

The first change, detailed in Figure 6, is fairly modest: the idea is simply to adopt the idea of computing and using a return address. The program is still in two parts, with a third part representing the stack:

1. Main is represented by the instructions held in addresses #0 . . . #14, and uses a, b and c held in addresses #15 . . . #17. A constant called off is held in address #18: this allows computation of the return address, representing the offset from where we copy the value of PC to where we should return.
2. Add is represented by the instructions held in addresses #19 . . . #23, and uses x, y and z held in addresses #24 . . . #26.
3. The stack occupies addresses #27 . . . #34; remember that it grows downward in memory, and that $SP = 34$ initially.

Notice the difference from the previous attempt: there we had a fixed return address (Add set $PC = 9$), but here we use the return address passed on the stack (Add sets PC equal to a value popped off the stack). This essentially solves the first problem we identified; Add can be called from wherever we want because as long as we push the right return address onto the stack, it will always know where to return. Adopting a similar abridged description as before, execution of the program can be described as follows:

- Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

$$\begin{array}{cccccccccccccc} SP & & & & & & & & & & & \downarrow & & \\ i & = & & \ldots, & 27, & 28, & 29, & 30, & 31, & 32, & 33, & 34, & \ldots & \\ \mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots & \rangle \end{array}$$

- Execution starts at address #0: Main first initialises a and b (addresses #0 . . . #3). It then computes and pushes the return address onto the stack (addresses #4 . . . #6): the instructions copy the value of PC, the add an offset to it giving the address #12, i.e., just after the instruction that performs the call to Add. Next it copies a and b into x and y (addresses #7 . . . #10).

  Main finally passes control to Add by setting $PC = 19$ (address #11). At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc} SP & & & & & & & & & & \downarrow & & & \\ i & = & & \ldots, & 27, & 28, & 29, & 30, & 31, & 32, & 33, & 34, & \ldots & \\ \mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 12, & \ldots & \rangle \\ & = & & & & & & & & & & RA & & \end{array}$$

- Add computes $z = x + y$ (addresses #19 . . . #21) and then pops the return address from the stack. Finally, it passes control to Main by copying the return address into PC. At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc} SP & & & & & & & & & & & \downarrow & & \\ i & = & & \ldots, & 27, & 28, & 29, & 30, & 31, & 32, & 33, & 34, & \ldots & \\ \mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 12, & \ldots & \rangle \end{array}$$

- Main concludes execution by copying z into c (addresses #12 . . . #13), then halting (address #14).

Address	Instruction			Purpose
0	200010	$\mapsto$	$\mathsf{A} \leftarrow 10$	compute $a = 10$
1	210015	$\mapsto$	$\mathsf{MEM}[15] \leftarrow \mathsf{A}$	store a
2	200020	$\mapsto$	$\mathsf{A} \leftarrow 20$	compute $b = 20$
3	210016	$\mapsto$	$\mathsf{MEM}[16] \leftarrow \mathsf{A}$	store b
4	440000	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{PC}$	copy PC
5	300018	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[18]$	compute $RA = PC + off$
6	500000	$\mapsto$	$\mathsf{MEM}[\mathsf{SP}] \leftarrow \mathsf{A}, \mathsf{SP} \leftarrow \mathsf{SP} - 1$	push RA
7	220015	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[15]$	load a
8	210024	$\mapsto$	$\mathsf{MEM}[24] \leftarrow \mathsf{A}$	store $x = a$
9	220016	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[16]$	load b
10	210025	$\mapsto$	$\mathsf{MEM}[25] \leftarrow \mathsf{A}$	store $y = b$
11	400019	$\mapsto$	$\mathsf{PC} \leftarrow 19$	call
12	220026	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[26]$	load z
13	210017	$\mapsto$	$\mathsf{MEM}[17] \leftarrow \mathsf{A}$	store $c = z$
14	100000	$\mapsto$	*HALT*	halt
15	000000	$\mapsto$	*NOP*	a
16	000000	$\mapsto$	*NOP*	b
17	000000	$\mapsto$	*NOP*	c
18	000007	$\mapsto$	*NOP*	off
19	220024	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[24]$	load x
20	300025	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[25]$	compute $z = x + y$
21	210026	$\mapsto$	$\mathsf{MEM}[26] \leftarrow \mathsf{A}$	store z
22	510000	$\mapsto$	$\mathsf{SP} \leftarrow \mathsf{SP} + 1, \mathsf{A} \leftarrow \mathsf{MEM}[\mathsf{SP}]$	pop RA
23	430000	$\mapsto$	$\mathsf{PC} \leftarrow \mathsf{A}$	return
24	000000	$\mapsto$	*NOP*	x
25	000000	$\mapsto$	*NOP*	y
26	000000	$\mapsto$	*NOP*	z
27	000000	$\mapsto$	*NOP*	stack
28	000000	$\mapsto$	*NOP*	stack
29	000000	$\mapsto$	*NOP*	stack
30	000000	$\mapsto$	*NOP*	stack
31	000000	$\mapsto$	*NOP*	stack
32	000000	$\mapsto$	*NOP*	stack
33	000000	$\mapsto$	*NOP*	stack
34	000000	$\mapsto$	*NOP*	stack (initial ToS)

Figure 6: *Attempt #2 at implementing* MAIN *and* ADD.

Address	Instruction			Purpose
0	200010	$\mapsto$	$\mathsf{A} \leftarrow 10$	compute $a = 10$
1	210015	$\mapsto$	$\mathsf{MEM}[15] \leftarrow \mathsf{A}$	store a
2	200020	$\mapsto$	$\mathsf{A} \leftarrow 20$	compute $b = 20$
3	210016	$\mapsto$	$\mathsf{MEM}[16] \leftarrow \mathsf{A}$	store b
4	440000	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{PC}$	copy PC
5	300018	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[18]$	compute $RA = PC + off$
6	500000	$\mapsto$	$\mathsf{MEM}[\mathsf{SP}] \leftarrow \mathsf{A}, \mathsf{SP} \leftarrow \mathsf{SP} - 1$	push RA
7	220015	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[15]$	load a
8	500000	$\mapsto$	$\mathsf{MEM}[\mathsf{SP}] \leftarrow \mathsf{A}, \mathsf{SP} \leftarrow \mathsf{SP} - 1$	push a
9	220016	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[16]$	load b
10	500000	$\mapsto$	$\mathsf{MEM}[\mathsf{SP}] \leftarrow \mathsf{A}, \mathsf{SP} \leftarrow \mathsf{SP} - 1$	push b
11	400019	$\mapsto$	$\mathsf{PC} \leftarrow 19$	call
12	510000	$\mapsto$	$\mathsf{SP} \leftarrow \mathsf{SP} + 1, \mathsf{A} \leftarrow \mathsf{MEM}[\mathsf{SP}]$	pop c
13	210017	$\mapsto$	$\mathsf{MEM}[17] \leftarrow \mathsf{A}$	store c
14	100000	$\mapsto$	*HALT*	halt
15	000000	$\mapsto$	*NOP*	a
16	000000	$\mapsto$	*NOP*	b
17	000000	$\mapsto$	*NOP*	c
18	000007	$\mapsto$	*NOP*	off
19	510000	$\mapsto$	$\mathsf{SP} \leftarrow \mathsf{SP} + 1, \mathsf{A} \leftarrow \mathsf{MEM}[\mathsf{SP}]$	pop b
20	210031	$\mapsto$	$\mathsf{MEM}[31] \leftarrow \mathsf{A}$	store y
21	510000	$\mapsto$	$\mathsf{SP} \leftarrow \mathsf{SP} + 1, \mathsf{A} \leftarrow \mathsf{MEM}[\mathsf{SP}]$	pop a
22	210030	$\mapsto$	$\mathsf{MEM}[30] \leftarrow \mathsf{A}$	store x
23	510000	$\mapsto$	$\mathsf{SP} \leftarrow \mathsf{SP} + 1, \mathsf{A} \leftarrow \mathsf{MEM}[\mathsf{SP}]$	pop RA
24	210032	$\mapsto$	$\mathsf{MEM}[32] \leftarrow \mathsf{A}$	store RA
25	220030	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[30]$	load x
26	300031	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[31]$	compute $z = x + y$
27	500000	$\mapsto$	$\mathsf{MEM}[\mathsf{SP}] \leftarrow \mathsf{A}, \mathsf{SP} \leftarrow \mathsf{SP} - 1$	push z
28	220032	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[32]$	load RA
29	430000	$\mapsto$	$\mathsf{PC} \leftarrow \mathsf{A}$	return
30	000000	$\mapsto$	*NOP*	x
31	000000	$\mapsto$	*NOP*	y
32	000000	$\mapsto$	*NOP*	RA
33	000000	$\mapsto$	*NOP*	stack
34	000000	$\mapsto$	*NOP*	stack
35	000000	$\mapsto$	*NOP*	stack
36	000000	$\mapsto$	*NOP*	stack
37	000000	$\mapsto$	*NOP*	stack
38	000000	$\mapsto$	*NOP*	stack
39	000000	$\mapsto$	*NOP*	stack
40	000000	$\mapsto$	*NOP*	stack (initial ToS)

Figure 7: *Attempt #3 at implementing* MAIN *and* ADD.

2.3.2 Attempt #3: passing arguments, returning values

The next change, detailed in Figure 7, is again fairly modest: since we already use the stack to house a return address, it makes sense to use it for the arguments and return value associated with the call to Add. The program is still in two parts, with a third part again representing the stack:

1. Main is represented by the instructions held in addresses #0 . . . #14, and uses a, b and c held in addresses #15 . . . #17; the constant off is held in address #18.
2. Add is represented by the instructions held in addresses #19 . . . #29, and uses x and y held in addresses #30 . . . #31. Notice that we no longer need any space for z (since we use the stack to hold it) but do need some space (address #32) to temporarily hold the return address.
3. The stack occupies addresses #33 . . . #40; remember that it grows downward in memory, and that $SP = 40$ initially.

The program execution can be described as follows:

- Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & & \downarrow & & \\
i & = & & \ldots, & 33, & 34, & 35, & 36, & 37, & 38, & 39, & 40, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots & \rangle
\end{array}$$

- Execution starts at address #0: Main first initialises a and b (addresses #0 . . . #3). It then computes and pushes the return address, a and b onto the stack (addresses #4 . . . #10).

  Main finally passes control to Add by setting $PC = 19$ (address #11). At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & \downarrow & & & & & \\
i & = & & \ldots, & 33, & 34, & 35, & 36, & 37, & 38, & 39, & 40, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 20, & 10, & 12, & \ldots & \rangle \\
 & = & & & & & & & & y & x & RA & &
\end{array}$$

- Add first pops b, a and the return address off the stack and stores them (addresses #19 . . . #24). It then computes $x + y$, pushing the result onto the stack rather than storing it in z as previously (addresses #25 . . . #27). Finally, it passes control to Main by loading the return address (previously popped off the stack and stored) and copying it into PC. At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & \downarrow & & & \\
i & = & & \ldots, & 33, & 34, & 35, & 36, & 37, & 38, & 39, & 40, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 20, & 10, & 30, & \ldots & \rangle \\
 & = & & & & & & & & & & z & &
\end{array}$$

- Main concludes execution by popping z off the stack and storing it into c (addresses #12 . . . #13). At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & & \downarrow & & \\
i & = & & \ldots, & 33, & 34, & 35, & 36, & 37, & 38, & 39, & 40, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 20, & 10, & 30, & \ldots & \rangle
\end{array}$$

  It then halts (address #14).

In a sense, the end result segregates Main and Add more than previously. No longer does Main access the local variables "owned" by Add, instead the two communicate with each other using the stack.

2.3.3 Attempt #4: avoiding individual pushes and pops

The length of (i.e., number of instructions in) the previous attempt is annoying. One of the causes is the large number of pushes and pops. The first thing Add does for example is pop x and y; since it has nowhere to keep them, it then has to store them back into memory which seems like a waste of time. The idea of our next attempt is to combine together individual pushes and pops, and to totally avoid them wherever possible.

For example, to push multiple items onto the stack the idea is to first update SP to make space for *all* the items in one go, then store the items relative to the new value of SP; this contrasts with updating SP for *each* item. Likewise instead of popping items off the stack and then having to store them somewhere, we access the items *on* the stack using the appropriate load and store instructions. Just as before, the program is in two parts with a third part representing the stack:

Address	Instruction			Purpose
0	200010	$\mapsto$	$\mathsf{A} \leftarrow 10$	compute $a = 10$
1	210017	$\mapsto$	$\mathsf{MEM}[17] \leftarrow \mathsf{A}$	store a
2	200020	$\mapsto$	$\mathsf{A} \leftarrow 20$	compute $b = 20$
3	210018	$\mapsto$	$\mathsf{MEM}[18] \leftarrow \mathsf{A}$	store b
4	580004	$\mapsto$	$\mathsf{SP} \leftarrow \mathsf{SP} - 4$	create stack frame
5	440000	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{PC}$	copy PC
6	300020	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[20]$	compute $RA = PC + off$
7	520004	$\mapsto$	$\mathsf{MEM}[\mathsf{SP} + 4] \leftarrow \mathsf{A}$	store RA
8	220017	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[17]$	load a
9	520003	$\mapsto$	$\mathsf{MEM}[\mathsf{SP} + 3] \leftarrow \mathsf{A}$	store a
10	220018	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[18]$	load b
11	520002	$\mapsto$	$\mathsf{MEM}[\mathsf{SP} + 2] \leftarrow \mathsf{A}$	store b
12	400021	$\mapsto$	$\mathsf{PC} \leftarrow 21$	call
13	530001	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[\mathsf{SP} + 1]$	load c
14	210019	$\mapsto$	$\mathsf{MEM}[19] \leftarrow \mathsf{A}$	store c
15	570004	$\mapsto$	$\mathsf{SP} \leftarrow \mathsf{SP} + 4$	remove stack frame
16	100000	$\mapsto$	*HALT*	halt
17	000000	$\mapsto$	*NOP*	a
18	000000	$\mapsto$	*NOP*	b
19	000000	$\mapsto$	*NOP*	c
20	000007	$\mapsto$	*NOP*	off
21	530003	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[\mathsf{SP} + 3]$	load x
22	540002	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{A} + \mathsf{MEM}[\mathsf{SP} + 2]$	compute $z = x + y$
23	520001	$\mapsto$	$\mathsf{MEM}[\mathsf{SP} + 1] \leftarrow \mathsf{A}$	store z
24	530004	$\mapsto$	$\mathsf{A} \leftarrow \mathsf{MEM}[\mathsf{SP} + 4]$	load RA
25	430000	$\mapsto$	$\mathsf{PC} \leftarrow \mathsf{A}$	return
26	000000	$\mapsto$	*NOP*	stack
27	000000	$\mapsto$	*NOP*	stack
28	000000	$\mapsto$	*NOP*	stack
29	000000	$\mapsto$	*NOP*	stack
30	000000	$\mapsto$	*NOP*	stack
31	000000	$\mapsto$	*NOP*	stack
32	000000	$\mapsto$	*NOP*	stack
33	000000	$\mapsto$	*NOP*	stack (initial ToS)

Figure 8: *Attempt #4 at implementing* MAIN *and* ADD.

1. MAIN is represented by the instructions held in addresses #0 . . . #16, and uses a, b and c held in addresses #17 . . . #19; the constant off is held in address #20.
2. ADD is represented by the instructions held in addresses #21 . . . #25. Notice that we no longer need any space for x, y nor z since they are all held on the stack.
3. The stack occupies addresses #26 . . . #33; remember that it grows downward in memory, and that $SP = 33$ initially.

The program execution can be described as follows:

- Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

$$\begin{array}{ccclrrrrrrrrll}
SP & & & & & & & & & & & \downarrow & & \\
i & = & & \ldots, & 26, & 27, & 28, & 29, & 30, & 31, & 32, & 33, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots & \rangle
\end{array}$$

- Execution starts at address #0: MAIN first initialises a and b (addresses #0 . . . #3). It then creates a stack frame (addresses #4) ready for the call to ADD: there is space for four items in the frame, namely the return address, x, y and the return value z. MAIN proceeds by computing and storing the return address, then the values of a and b into the stack frame (addresses #5 . . . #11) to form the arguments x and y.

  Finally it passes control to ADD by setting $PC = 21$ (address #12). At this point, the memory content is as follows:

$$\begin{array}{ccclrrrrrrrrll}
SP & & & & & & & \downarrow & & & & & & \\
i & = & & \ldots, & 26, & 27, & 28, & 29, & 30, & 31, & 32, & 33, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 20, & 10, & 13, & \ldots & \rangle \\
 & = & & & & & & & z & y & x & RA & &
\end{array}$$

- ADD no longer needs to pop x and y; it simply loads the values from the stack frame, computes $z = x + y$ and then stores the result back into the stack frame (addresses #21 . . . #23). At this point, the memory content is as follows:

$$\begin{array}{ccclrrrrrrrrll}
SP & & & & & & & \downarrow & & & & & & \\
i & = & & \ldots, & 26, & 27, & 28, & 29, & 30, & 31, & 32, & 33, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 30, & 20, & 10, & 13, & \ldots & \rangle \\
 & = & & & & & & & z & y & x & RA & &
\end{array}$$

  Finally, it passes control to MAIN by loading the return address from the stack frame and copying it into PC (addresses #24 . . . #25).

- MAIN concludes execution by loading z from the stack frame and storing it into c (addresses #13 . . . #14); it then removes the stack frame (addresses #15). At this point, the memory content is as follows:

$$\begin{array}{ccclrrrrrrrrll}
SP & & & & & & & & & & & \downarrow & & \\
i & = & & \ldots, & 26, & 27, & 28, & 29, & 30, & 31, & 32, & 33, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 30, & 20, & 10, & 13, & \ldots & \rangle
\end{array}$$

  It then halts (address #14).

Notice that now ADD in particular is more efficient: it no longer has to pop then store x and y since it can simply access them using the appropriate offset from SP. The act of MAIN creating and then removing the stack frame for ADD is also more explicit since it happens in one step rather than within a series of pushes and pops.

2.3.4 Attempt #5: using the stack frame to house local variables

Now we are fairly close to what we want; the only lingering issue relates to the second problem we originally identified. Look at the previous attempt: MAIN has a single place in memory it can store a, b and c. In this case MAIN is not recursive so there is no major problem, but we need a solution for more general cases.

So the question is, where *can* we store local variables? The answer is to pull the same trick again, and simply store them on the stack: if each instance of a sub-routine has a dedicated stack frame, then storing local variables in the stack frame will mean they cannot be overwritten by some other instance. In the last attempt, the caller took the responsibility of manging the stack frame. In this attempt the callee will also play a part:

1. MAIN is represented by the instructions held in addresses #0 . . . #18. Notice that we no longer need any space for a, b nor c since they are all held on the stack; off is a constant (we never change it) so that does not need to be held on the stack.

Address	Instruction			Purpose
0	580004	↦	SP ← SP − 4	create local space
1	200010	↦	A ← 10	compute $a = 10$
2	520004	↦	MEM[SP + 4] ← A	store a
3	200020	↦	A ← 20	compute $b = 20$
4	520003	↦	MEM[SP + 3] ← A	store b
5	580004	↦	SP ← SP − 4	create stack frame
6	440000	↦	A ← PC	copy PC
7	300019	↦	A ← A + MEM[19]	compute $RA = PC + off$
8	520004	↦	MEM[SP + 4] ← A	store RA
9	530008	↦	A ← MEM[SP + 8]	load a
10	520003	↦	MEM[SP + 3] ← A	store a
11	530007	↦	A ← MEM[SP + 7]	load b
12	520002	↦	MEM[SP + 2] ← A	store b
13	400020	↦	PC ← 20	call
14	530001	↦	A ← MEM[SP + 1]	load c
15	520006	↦	MEM[SP + 6] ← A	store c
16	570004	↦	SP ← SP + 4	remove stack frame
17	570004	↦	SP ← SP + 4	remove local space
18	100000	↦	*HALT*	halt
19	000007	↦	*NOP*	off
20	530003	↦	A ← MEM[SP + 3]	load x
21	540002	↦	A ← A + MEM[SP + 2]	compute $z = x + y$
22	520001	↦	MEM[SP + 1] ← A	store z
23	530004	↦	A ← MEM[SP + 4]	load RA
24	430000	↦	PC ← A	return
25	000000	↦	*NOP*	stack
26	000000	↦	*NOP*	stack
27	000000	↦	*NOP*	stack
28	000000	↦	*NOP*	stack
29	000000	↦	*NOP*	stack
30	000000	↦	*NOP*	stack
31	000000	↦	*NOP*	stack
32	000000	↦	*NOP*	stack (initial ToS)

Figure 9: *Attempt #5 at implementing* MAIN *and* ADD.

2. ADD is represented by the instructions held in addresses #20 . . . #24.
3. The stack occupies addresses #25 . . . #32; remember that it grows downward in memory, and that $SP = 32$ initially.

The program execution can be described as follows:

- Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

$$\begin{array}{cccccccccccccc} SP & & & & & & & & & & & \downarrow & & \\ i & = & & \ldots, & 25, & 26, & 27, & 28, & 29, & 30, & 31, & 32, & \ldots & \\ \textsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots & \rangle \end{array}$$

- Execution starts at address #0: the first thing MAIN now does is create space in the stack frame for local variables (address #0). There is space for three items in the frame, namely a, b and c. Next it initialises a and b (addresses #1 . . . #4) and creates a stack frame (addresses #5) ready for the call to ADD. MAIN proceeds by computing and storing the return address, then a and b into the stack frame (addresses #6 . . . #12).

  Finally it passes control to ADD by setting $PC = 20$ (address #13). At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc} SP & & & & \downarrow & & & & & & & & & \\ i & = & & \ldots, & 25, & 26, & 27, & 28, & 29, & 30, & 31, & 32, & \ldots & \\ \textsf{MEM} & = & \langle & \ldots, & 0, & 0, & 20, & 10, & 14, & 0, & 20, & 10, & \ldots & \rangle \\ & = & & & & z & y & x & RA & c & b & a & & \end{array}$$

- ADD is basically the same as the last attempt: it loads x and y from the stack frame, computes $z = x + y$ and then stores the result back into the stack frame (addresses #20 . . . #22). At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc} SP & & & & \downarrow & & & & & & & & & \\ i & = & & \ldots, & 25, & 26, & 27, & 28, & 29, & 30, & 31, & 32, & \ldots & \\ \textsf{MEM} & = & \langle & \ldots, & 0, & 30, & 20, & 10, & 14, & 0, & 20, & 10, & \ldots & \rangle \\ & = & & & & z & y & x & RA & c & b & a & & \end{array}$$

  Finally, it passes control to MAIN by loading the return address from the stack frame and copying it into PC (addresses #23 . . . #24).

- MAIN concludes execution by loading z from the stack frame and storing it into c (addresses #14 . . . #15); it then removes the stack frame (addresses #16) to give

$$\begin{array}{cccccccccccccc} SP & & & & & & & & \downarrow & & & & & \\ i & = & & \ldots, & 25, & 26, & 27, & 28, & 29, & 30, & 31, & 32, & \ldots & \\ \textsf{MEM} & = & \langle & \ldots, & 0, & 30, & 20, & 10, & 14, & 30, & 20, & 10, & \ldots & \rangle \\ & = & & & & & & & & c & b & a & & \end{array}$$

  and the space for local variables (addresses #17) to give

$$\begin{array}{cccccccccccccc} SP & & & & & & & & & & & \downarrow & & \\ i & = & & \ldots, & 25, & 26, & 27, & 28, & 29, & 30, & 31, & 32, & \ldots & \\ \textsf{MEM} & = & \langle & \ldots, & 0, & 30, & 20, & 10, & 14, & 30, & 20, & 10, & \ldots & \rangle \end{array}$$

  It then halts (address #18).

At this point we are done: the end result is more or less a real sub-routine calling mechanism. Of course there are aspects that could be improved (some depend on the instruction set we are using), but this really is how things work.

3 A buffer overflow attack

Now we finally have enough of a background to look at the original problem: buffer overflow attacks. The attack is based on the fact that the stack holds a mix of return addresses (which determine control-flow) and local variables (which are used in computation). The basic idea is that if we put too much content into the local variables, the overwriting bug is triggered; what we overwrite is essentially other stack content, including return addresses. That is only ever going to be bad, basically because it breaks our sub-routine calling mechanism.

```
algorithm WEB-SERVER begin
    forever do
        PROCESS-REQ()
    end
end
```

(a) *An algorithm that models a web-server.*

```
algorithm PROCESS-REQ begin
    $T \leftarrow \langle \perp, \perp \rangle$
    $i \leftarrow 0$
    repeat
        $t \leftarrow$ NETWORK-READ()
        if $t \neq 0$ then
            $T_i \leftarrow t$
            $i \leftarrow i + 1$
        end
    until $t = 0$
    Process $T$ somehow return
end
```

(b) *An algorithm that models the reading and processing of some request.*

Figure 10: *Two algorithms used to represent the web-server subjected to a buffer overflow attack.*

3.1 Modelling and implementing a "web-server"

First we need something to attack, so we return to our original motivating example of a web-server. The scenario is that a remote computer C acts as a target web-server, and our task as the attacker Eve is to send it a malicious request S; the request is specifically constructed to make C do whatever *we* want, rather than whatever it was designed to. Figure 10 describes WEB-SERVER and PROCESS-REQ, two algorithms that act as a model for C:

- The WEB-SERVER algorithm simply loops continuously; this models the fact that C endlessly listens for a connection by some web-browser, then services the request it makes.

- The PROCESS-REQ algorithm is where the action happens; it models C reading a request and then processing it somehow. The request itself is a sequence of numbers terminated by a zero; this is a bit like the C-string method from Chapter 5.

  Imagine that we send the request

  $$S = \langle 1, 2, 0 \rangle.$$

  The web-server C processes the request by following the PROCESS-REQ algorithm as follows:

  Step #1 Assign $T \leftarrow \langle \perp, \perp \rangle$, $i \leftarrow 0$.

  Step #2 Assign $t \leftarrow$ NETWORK-RD, i.e., read the value 1 from the network.

  Step #3 Since $t \neq 0$, assign $T_0 \leftarrow t = 1$ and $i \leftarrow i + 1 = 1$,

  Step #4 Since $t \neq 0$, perform the next loop iteration.

  Step #5 Assign $t \leftarrow$ NETWORK-RD, i.e., read the value 2 from the network.

  Step #6 Since $t \neq 0$, assign $T_1 \leftarrow t = 2$ and $i \leftarrow i + 1 = 2$,

  Step #7 Since $t \neq 0$, perform the next loop iteration.

  Step #8 Assign $t \leftarrow$ NETWORK-RD, i.e., read the value 0 from the network.

  Step #9 Since $t = 0$, skip the conditional block.

  Step #10 Since $t = 0$, stop the loop.

  Step #11 Process T somehow.

  Step #12 Return.

  That is, it reads each element of the request (apart from the zero which terminates it) into the sequence T, then processes T in some way. The type of processing performed is not important; you can think of it being interpreted as as some request for a particular web-page if you like.

  The very first step highlights a problem however: how do we know the size of T? Of course we cannot know the size until we have read it, so we simply allocate a 2-element buffer and hope this is enough (i.e., that the request is not longer).

Address	Instruction			Purpose
0	580001	↦	SP ← SP − 1	create stack frame
1	440000	↦	A ← PC	copy *PC*
2	300007	↦	A ← A + MEM[7]	compute $RA = PC + off$
3	520001	↦	MEM[SP + 1] ← A	store *RA*
4	400008	↦	PC ← 8	call
5	570001	↦	SP ← SP + 1	remove stack frame
6	400000	↦	PC ← 0	loop
7	000003	↦	*NOP*	off
8	580002	↦	SP ← SP − 2	create local space
9	120000	↦	read A from network port 0	read from network
10	410016	↦	PC ← 16 iff. A = 0	check *t*
11	520001	↦	MEM[SP + 1] ← A	store *t* into T_i
12	220011	↦	A ← MEM[11]	load "*T* access" instruction
13	300019	↦	A ← A + MEM[19]	compute "*T* access" instruction
14	210011	↦	MEM[11] ← A	store "*T* access" instruction
15	400009	↦	PC ← 9	loop
16	570002	↦	SP ← SP + 2	remove local space
17	530001	↦	A ← MEM[SP + 1]	load *RA*
18	430000	↦	PC ← A	return
19	000001	↦	*NOP*	*inc*
20	000000	↦	*NOP*	stack
21	000000	↦	*NOP*	stack
22	000000	↦	*NOP*	stack
23	000000	↦	*NOP*	stack
24	000000	↦	*NOP*	stack
25	000000	↦	*NOP*	stack
26	000000	↦	*NOP*	stack (initial ToS)

Figure 11: *A program that implements the* Web-Server *and* Process-Req *algorithms.*

Remember this is *just* a model: the claim is not that this is a *real* web-server, just that we have an outline of the parts which are important to our discussion. Put another way, this is a compromise between reality and the ability to explain the main points clearly.

Now we need to implement these algorithms in a program; this means another upgrade for our example computer that adds network access. This sounds like (and would be) a complicated task if it were not for the fact that we can avoid almost all the detail involved: all we want is some instructions that can read and write values, acting as a model for what a real network connection would do. In short, we add the following:

- 12*nnnn* means read A from network port *n*.
- 13*nnnn* means write A onto network port *n*.

This means that we, as the attacker, can supply numbers which are read using the first instruction and view numbers written using the second instruction. Armed with the new instructions, Figure 11 describes the implementation. Following the same style as throughout this Chapter, the program is in two parts, with a third part representing the stack:

1. Web-Server is represented by the instructions held in addresses #0 . . . #6. A constant called off is held in address #7: this allows computation of the return address.
2. Process-Req is represented by the instructions held in addresses #8 . . . #18. A constant called *inc* is held in address #19: this allows addition of a fixed value (in this case one) to a value of our choice.

   Unlike most of the previous programs we have looked at before, this one uses one fairly opaque technique to get the job done. The problem is, how can we store *t* into T_i? Since *T* is a local variable, it is held on the stack; ideally we would do something like

   $$\text{MEM} \leftarrow A$$

   This would allow us to read *t* into *A*, then store it at the right offset from *SP*. But we cannot do this; there is no appropriate instruction. So instead, we "make" one ourselves. The idea is to make constructive use of the self-modifying code from Chapter 4. Notice that the instruction held in address #11 initially reads

   510001 ↦ MEM.

How can we make it access the next element of the stack (i.e., the next element of T)? Easy! Just add one to it; we get

$$510002 \mapsto \mathsf{MEM}.$$

So in each iteration of the loop, we load the instruction and update it so that in the next iteration it will store into the next element of the stack; this is achieved by the instructions held in addresses #12 ... #14. Put another way, we do not need i: we just have an instruction and make sure the operand is updated to "point" at where i would.

3. The stack occupies addresses #19 ... #27; remember that it grows downward in memory, and that $SP = 27$ initially.

3.2 A normal, valid request

When we feed the web-server a normal request, things proceed as we would expect. Imagine we go back to the previous example and feed our program the input

$$S = \langle 1, 2, 0 \rangle.$$

The behaviour of the program is as follows:

- Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & & \downarrow & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots & \rangle
\end{array}$$

- Execution starts at address #0: Web-Server creates a stack frame (addresses #0) then computes and stores a return address (addresses #1 ... #3) in it. Then it passes control to Process-Req by setting $PC = 8$ (address #4). At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & \downarrow & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 5, & \ldots & \rangle \\
 & = & & & & & & & & & & RA & &
\end{array}$$

- Process-Req first creates space in the stack frame for local variables (address #8); there is space for the 2-element buffer T. At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & \downarrow & & & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 5, & \ldots & \rangle \\
 & = & & & & & & & & T_0 & T_1 & RA & &
\end{array}$$

  It then loads t from the network (address #9), checks to see if this means the end of the sequence (address #10) and if not, stores t in T_i (address #11). In this case $t = 1$ since this is the first element of input, so at this point the memory content is:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & \downarrow & & & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 1, & 0, & 5, & \ldots & \rangle \\
 & = & & & & & & & & T_0 & T_1 & RA & &
\end{array}$$

  One more element is read from the network, which means the memory content once T is ready for processing is

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & \downarrow & & & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 1, & 2, & 5, & \ldots & \rangle \\
 & = & & & & & & & & T_0 & T_1 & RA & &
\end{array}$$

  After processing T (which is not included in the program), the space for local variables is removed (addresses #16) to give

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & \downarrow & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 1, & 2, & 5, & \ldots & \rangle \\
 & = & & & & & & & & & & RA & &
\end{array}$$

and control is passed back to Web-Server by loading the return address from the stack frame and copying it into PC (addresses #17 . . . #18).

- Main removes the stack frame (addresses #16) to give

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & & \downarrow & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 1, & 2, & 5, & \ldots & \rangle
\end{array}$$

  then sets $PC = 0$ (address #6) to repeat the whole process all over again.

3.3 A not-so-normal, attack request

As you might have guessed, we can feed the web-server a not-so-normal request and cause it problems. This time imagine we feed the program the input

$$S = \langle 111111, 111111, 25, 0\rangle.$$

Already this looks quite bad because the sequence has three elements in it (excluding the zero), and we only have enough space for two elements in the buffer for T. So what happens? This does:

- Initially, i.e., before the program starts to execute, the stack is empty; the content can be described as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & & \downarrow & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & \ldots & \rangle
\end{array}$$

- Execution starts at address #0: Web-Server creates a stack frame (addresses #0) then computes and stores a return address (addresses #1 . . . #3) in it. Then it passes control to Process-Req by setting $PC = 8$ (address #4). At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & \downarrow & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 5, & \ldots & \rangle \\
 & = & & & & & & & & & & RA & &
\end{array}$$

- Process-Req first creates space in the stack frame for local variables (address #8); there is space for the 2-element buffer T. At this point, the memory content is as follows:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & \downarrow & & & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 0, & 0, & 5, & \ldots & \rangle \\
 & = & & & & & & & & T_0 & T_1 & RA & &
\end{array}$$

  It then loads t from the network (address #9), checks to see if this means the end of the sequence (address #10) and if not, stores t in T_i (address #11). In this case $t = 111111$ since this is the first element of input, so at this point the memory content is:

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & \downarrow & & & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 111111, & 0, & 5, & \ldots & \rangle \\
 & = & & & & & & & & T_0 & T_1 & RA & &
\end{array}$$

  Two more elements are read from the network, which means the memory content once T is ready for processing is

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & \downarrow & & & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 111111, & 111111, & 25, & \ldots & \rangle \\
 & = & & & & & & & & T_0 & T_1 & RA & &
\end{array}$$

  Notice that the overwriting bug has been triggered. After processing T (which is not included in the program), the space for local variables is removed (addresses #16) to give

$$\begin{array}{cccccccccccccc}
SP & & & & & & & & & & \downarrow & & & \\
i & = & & \ldots, & 20, & 21, & 22, & 23, & 24, & 25, & 26, & 27, & \ldots & \\
\mathsf{MEM} & = & \langle & \ldots, & 0, & 0, & 0, & 0, & 0, & 111111, & 111111, & 25, & \ldots & \rangle \\
 & = & & & & & & & & & & RA & &
\end{array}$$

and an *attempt* to pass control to Web-Server is made. The return address is loaded from the stack frame and copied into *PC* (addresses #17 . . . #18).

But, since we read more data into *T* than it could hold, we overwrote the return address: this means that *PC* is set to 25 which we (as the attacker) provided. Instead of jumping back to address #5, i.e., returning to Web-Server, we instead end up executing the instruction at address #25. What is this instruction? None other than

$$111111 \mapsto \textbf{payload}$$

which we (as the attacker) provided, and does something "bad".

3.4 The end result

We started with an abstract example of the overwriting bug, and now we have a more concrete example of what it implies: the buffer overflow attack allows us to send the web-server some input, and have it execute a program (i.e., one or more instructions) of our choice. As we mentioned previously, this is bad. The things to note are that we

- needed no access to the web-server in the sense of having to log in and execute our program (meaning we side-step any password),
- needed no physical access to the web-server in the sense that the attack can be carried out across a network (from another country if we want), and
- tricked the web-server to execute our program as itself, meaning our program can enjoy access to any resource the web-server has.

Just like at the end of Chapter 4 where we tried to come up with better approaches to stopping viruses, we can try to think of ways to stop the buffer overflow attack. There are a lot of options, so as an exercise try to think of some: imagine you have free reign to alter the computer or the web-server program, and try to work out countermeasures that prevent our attack from working.

Research (task #2)

This is an ideal moment to point out that the impact of buffer overflow attacks (and related techniques) mean this is an active field of research; **black hat** [1] and **white hat** [16] researchers both push the field forward. In addition to our normal set of references to Wikipedia, a great online resource is **Phrack** [11] magazine. Some people might argue some of the content is morally dubious, but you can (mostly) ignore them: Phrack was, and still is, both an outstanding technical resource and documentary of information security and hacker [8] culture. You can find back-issues of Phrack here

`http://www.phrack.com/`

and one of the first articles on buffer overflow attacks here

`http://www.phrack.com/issues.html?issue=49&id=14`

The article has a lot more technical detail than what we have covered, and applies to more realistic scenarios, but basically the concept is exactly the same: go and read it!

References

[1] *Wikipedia: Black hat*. `http://en.wikipedia.org/wiki/Black_hat` (see p. 24).

[2] *Wikipedia: Bounds checking*. `http://en.wikipedia.org/wiki/Bounds_checking` (see p. 3).

[3] *Wikipedia: Buffer overflow*. `http://en.wikipedia.org/wiki/Buffer_overflow` (see p. 3).

[4] *Wikipedia: Call stack*. `http://en.wikipedia.org/wiki/Call_stack` (see p. 12).

[5] *Wikipedia: Droste effect*. `http://en.wikipedia.org/wiki/Droste_effect` (see p. 6).

[6] *Wikipedia: Fibonacci number*. `http://en.wikipedia.org/wiki/Fibonacci_number` (see p. 6).

[7] *Wikipedia: Function*. `http://en.wikipedia.org/wiki/Function_(computer_science)` (see p. 5).

[8] *Wikipedia: Hacker*. `http://en.wikipedia.org/wiki/Hacker_(computer_security)` (see p. 24).

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

[10] *Wikipedia: My Cup Runneth Over*. http://en.wikipedia.org/wiki/My_Cup_Runneth_Over (see p. 3).

[11] *Wikipedia: Phrack*. http://en.wikipedia.org/wiki/Phrack (see p. 24).

[12] *Wikipedia: Privilege level*. http://en.wikipedia.org/wiki/Privilege_level (see p. 4).

[13] *Wikipedia: Recursion*. http://en.wikipedia.org/wiki/Recursion (see p. 6).

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

[15] *Wikipedia: Stack*. http://en.wikipedia.org/wiki/Stack_(data_structure) (see p. 9).

[16] *Wikipedia: White hat*. http://en.wikipedia.org/wiki/White_hat (see p. 24).

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!

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.