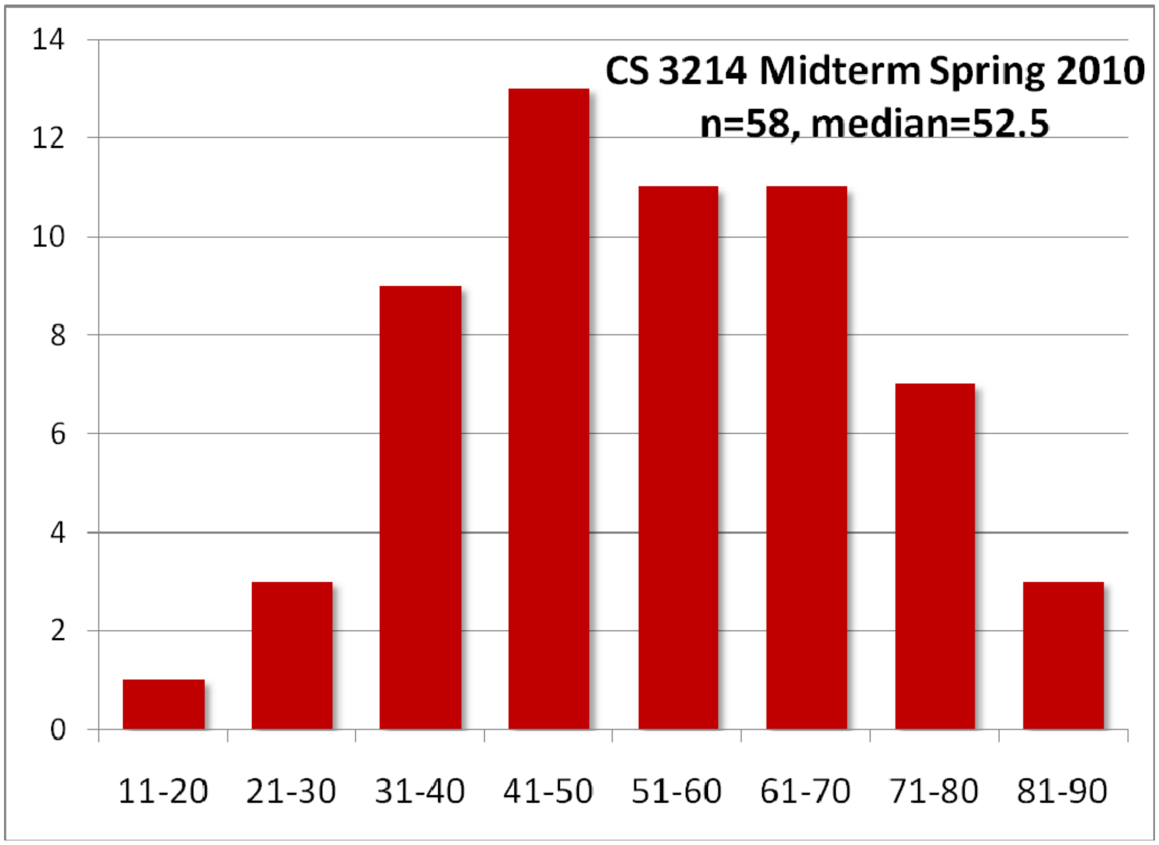
CS 3214 Midterm Solution

58 students took the midterm. The table below shows for each problem who graded it, as well as statistics about each problem. If you have a question about your score, please contact the person who graded the problem first before contacting your instructor.

	Problem 1	Problem 2	Problem 3	Problem 4	Total
Min	1	0	6	0	20
Max	30	29	26	12	88
Possible	30	30	28	12	100
Avg	15.8	13.3	16.7	7.6	53.4
StDev	7.2	6.9	5.2	2.5	16.0
Median	16	13	17	6	52.5
Grader	Scott: a) b) Godmar: c) d)	Peter	Ali	Godmar	



Solutions are shown in this style.
Grading comments are shown in this style.

1. Executing Programs on IA32 (30 pts)

The following questions relate to how programs are compiled for IA32.

a) (8 pts) Consider the following buggy program contained in a file sum.c

```
#include <stdio.h>

int
sum(int a, int b)
{
    int s = a + b;
    // return statement is missing
}

int
main()
{
    printf("%d\n", sum(1, 2));
}
```

i. (4 pts) When the program is compiled with 'gcc –o sum sum.c' and run, it will output '3'. Explain why 3 is output!

Since no optimization level is specified, the compiler will emit code for all statements it sees. This includes the computation of 's' as the sum of 'a' and 'b'. 's' happens to be computed in register $eax, so it coincidentally becomes the return value of sum.

ii. (4 pts) When the program is compiled with 'gcc –O2 –o sum sum.c' and run, it outputs a number such as -1074516556. Which analysis or optimization on the part of the compiler causes the generation of code that leads to this different result?

The compiler determines that 's' is not used and thus does not emit code to compute it - $eax in this case contains whichever value it had from the last time it was used. Or, if this function is inlined, the compiler may completely remove the function call since none of its computed values is used, then pass an uninitialized value to printf() (essentially, whatever is in memory at the address where printf expects its second argument).

b) (12 pts) Consider the following assembly code, which was produced by gcc for a function 't'. The left column shows the result when compiling with optimizations at level 1 (-O1).

IA 32 Code,compiled with –O1	C Code

`t:`
`    pushl   %ebp`
	void t(struct node *node)

```
    movl    %esp, %ebp
    pushl   %ebx
    subl    $4, %esp
    movl    8(%ebp), %ebx
    movl    (%ebx), %eax
    testl   %eax, %eax
    je  .L2
    movl    %eax, (%esp)
    call    t
.L2:
    movl    %ebx, (%esp)
    call    visit
    movl    4(%ebx), %eax
    testl   %eax, %eax
    je  .L6
    movl    %eax, (%esp)
    call    t
.L6:
    addl    $4, %esp
    popl    %ebx
    popl    %ebp
    ret
```

```
{
  if (node->left)
    t(node->left);
  visit(node);
  if (node->right)
    t(node->right);
}
```

Provide a C version of function t()!

Hint: t() accepts a pointer to this struct:

```
struct node {
   struct node *left;
   struct node *right;
}
```

c) (5 pts) Consider the following program:

```
#include <stdio.h>

int
mystery_function(int arg, ...)
{
    int * p = &arg;
    int   r = 1;

    while (*p)
        r = r * *p++;

    return r;
}

int
main()
{
    printf("%d\n", mystery_function(5, 4, 3, 2, 1, 0));
}
```

What does this program output when run and compiled under gcc on IA32?

The function iterates over its arguments and multiplies them until it finds a 0, so the output is 120. (Note that this code is not portable - use stdarg.h for portable code as done in the exercise.)

d) (5 pts) Consider how compilers generate code for switch() statements in C. Under which circumstances would a chain of if/else statements (shown left) yield better performance than a switch statement (shown right)?

```
if (x == CHOICE1)
  f1();
else if (x == CHOICE2)
  f2();
else if (x == CHOICE3)
  f3();
else
  . . .
```

```
switch (x) {
case CHOICE1:
     f1(); break;
case CHOICE2:
     f2(); break;
case CHOICE3:
     f3(); break;
. . .
}
```

For the if-else chain to outperform the switch statement, the likelihood of x being CHOICE1 must be much larger than the likelihood of x being CHOICE2, and so on. In addition, it should be true that there are either few CHOICE values, or a sparse distribution of CHOICE values, thus preventing the compiler from creating a jump table.

To obtain full credit, you needed to realize that the answer depended on the likelihood of 'x' values. Without making any assumptions about their likelihood, an if-else chain is never better than a switch, because the compiler will find an arrangement for the switch that provides good average and worst case performance (be it a if-else chain for a small number of choices, or a jump table for many dense choices, or a binary search tree for many sparse choices). What the compiler cannot optimize for is best-case performance, but a carefully arranged if-else chains – unless the compiler can create a jump table, in which case it can achieve constant time performance for all possible values of x.

Some implied that there would be a runtime cost for the jump table – this is not so, the table is created at compile time. (There is a space cost, but this cost rarely matters for performance.)

2. Structs and Arrays (30 pts)

a) (3x5 pts) The following 3 functions operate either on nested or on multi-level arrays of integers. Provide C versions of each function, filling in the missing parameter and body:

`access1:`
`    pushl   %ebp`
`    movl    %esp, %ebp`
`    imull   $120, 12(%ebp), %edx`
`    movl    16(%ebp), %eax`
`    leal    (%edx,%eax,4), %eax`
`    movl    8(%ebp), %edx`
`    movl    (%eax,%edx), %eax`
`    popl    %ebp`
`    ret`
	`int access1(`int A[][30]`, int x, int y)`
	`{`
	return A[x][y];
	`}`

`access2:`
`    pushl   %ebp`
`    movl    %esp, %ebp`
`    imull   $120, 12(%ebp), %eax`
`    addl    8(%ebp), %eax`
`    popl    %ebp`
`    ret`
	`int * access2(`int A[][30]`, int x)`
	`{`
	return A[x];
	`}`

`access3:`
`    pushl   %ebp`
`    movl    %esp, %ebp`
`    movl    12(%ebp), %edx`
`    movl    8(%ebp), %eax`
`    movl    (%eax,%edx,4), %edx`
`    movl    16(%ebp), %eax`
`    movl    (%edx,%eax,4), %eax`
`    popl    %ebp`
`    ret`
	`int access3(`int *A[]`, int x, int y)`
	`{`
	return A[x][y];
	`}`

The highest dimension doesn't matter, so you could have written int A[anything][30], it doesn't change what code is generated.

b) (15 pts) Consider the following piece of code:

```
#include <stdio.h>
struct alpha {
      int x;
      char a;
      int z;
      char b;
      char c;
      char d;
};
int main()
{
      int data[] = {8, 49, 0, 7, 4, 53, 6, 1, 0};
      struct alpha* ap = (struct alpha *) data;
      printf ("%d %c %d, %d %c %d\n", ap[0].x, ap[0].a, ap[0].z,
                                      ap[1].x, ap[1].a, ap[1].z);
}
```

i. (5 pts) Draw a picture of how the array “`data`” is laid out in memory. On the same picture, indicate where all the elements of `ap[0]` and `ap[1]`are stored.

data	8 0 0 0	49 0 0 0	0 0 0 0	7 0 0 0	4 0 0 0	53 0 0 0	6 0 0 0	1 0 0 0	0 0 0 0
	ap[0].x	ap[0].a	ap[0].z	.b .c .d	ap[1].x	ap[1].a	ap[1].z	.b .c .d	

ii. (3 pts) What is the output of the above program?
Hint: ASCII code for character ‘0’ is 48.

8 1 0, 4 5 6

iii. (3 pts) How would you rearrange the elements of `alpha` so as to minimize the amount of memory used to store it? Provide C code!

Any arrangement that groups the ‘char’ fields adjacent to each other works, such as:

```
struct alpha {
      int x;
      int z;
      char a;
      char b;
      char c;
      char d;
};
```

The size of the struct is reduced from 16 to 12.

iv. (4 pts) What will be the program output with your new version of `alpha`?

For above code rearrangement,
8 49, 7 5 4

(Note that outputting a zero byte value (‘\0’) using %c will not produce any output, nor advance the terminal’s cursor. You can see the 0 byte by piping the output to the ‘od’ command.)

We did not expect that you knew how a Unix terminal handles outputting ‘\0’.

3. Buffer Overflows (28 pts)

In project 2, we explored how the lack of buffer bounds checks can allow a malicious attacker to exploit a program vulnerability and execute dangerous code. In this question, we will explore several techniques used to deter such buffer overflow attacks.

a) (4 points) A proposed technique involves programming the memory protection hardware to disallow execution of instructions fetched from stack memory. Explain the rationale for this technique!

Buffer overflow attacks require that code provided by the attacker be executed. If the overflowing buffer is located on the stack, the exploit code will be written there. Not allowing execution of code located on the stack will thwart this type of attack, even if the attacker succeeds in placing exploit code on the stack.

b) Another approach to avoiding buffer overflows is to place a random "canary" word below the return address on the stack. A canary word is a special value that, if changed, indicates that the return address may have been compromised. Thus, if the return address is changed via a buffer overflow, the canary is destroyed and the system can detect the attack.

i. (4 points) Why is it difficult for the attack code to compromise the return address while leaving the canary intact?

gets() will place the exploit string consecutively in memory – since the return address is located at a higher address than the canary word, the canary word is destroyed if the return address is.

ii. (4 points) How would the code of the function, which the compiler creates, need to be changed to make use of the canary?

On procedure entry, the canary word must be stored. When a return instruction is executed, the canary word must be read and checked against the expected value.

iii. (4 points) Consider the following code:

```
void good_func (void)
{
   printf("This is the good function\n");
}

void bad_func (void)
{
   printf("This is the bad function\n");
}

void func_caller (void (*func_ptr)(void))
{
   char buff[10];
   gets(buff); /* get input from user via unsafe func */
   func_ptr(); /* call function pointed to by func_ptr */
}

int main()
{
   func_caller (good_func);
}
```

In this code, `main` calls `func_caller` with a pointer `good_func`, which then invokes this function using the passed pointer. If gets() reads less than 10 bytes, the output of the program is "`This is the good function`". Based on your answer to part ii), discuss if the use of a canary word could prevent an attacker from crafting an exploit string that would cause the invocation of `bad_func` instead!

No it would not since the indirect function call occurs before the 'ret' instruction is reached.

(If your answer to b) included the assumption that the compiler generates code that checks the canary before each indirect function call as well as before returning, the answer would be yes, because 'func_ptr' is located above the canary on the stack.)

c) (4 points) Could the vulnerability described in part b) be avoided using address space randomization, which is a technique that places a program's stack at randomly chosen addresses, which differ from run to run?

The attack described in part b) iii) cannot, because it does not rely on knowing the address of the current stack frame – unlike a conventional buffer overflow attack in which the return address is overwritten to point to exploit code located on the stack.
However, a generic stack overflow attack (such as the one you implemented in Project 2), often can be prevented.
The question was vague by referring to 'part b)', so both answers were accepted if the context was made clear.

d) Assume that a program contains the following bug:

```
if (some rare error condition occurs) {
      int errorcode = 31;
      char errormsg[] = "Some Rare Error Occurred\n";
      printf("Error: %d: %x", errorcode, errormsg);
}
```

The correct format string would have been "`Error: %d: %s`".

i. (4 pts) Is this bug exploitable, i.e., could an attacker use it to achieve the execution of code under their control? Justify your answer!

No, this bug does not allow the injection of code controlled by an attacker.

ii. (4 pts) Is this bug security-relevant, i.e., could it aid or amplify a possible attack? Justify your answer!

Yes, if an attacker can trigger the ‘rare error condition,’ they will learn the address of ‘errormsg[0]’, which is an address within the current stack frame. This knowledge can help craft an exploit if there is a (separate) buffer overflow vulnerability.

4. Optimization (12 pts)

Consider the following 2 versions of a function that computes the square root of a number using Newton’s algorithm. The square root is written to the pointer referred to by ‘root’, and the number of iterations is returned. Both versions are nearly identical, except that the second version uses a local variable ‘x’ to hold the intermediate values of the to-be-computed square root.

```
int square_root_newton(double n, double *root)
{
  double xn = 1.0;
  int niter = 0;

  do {
      *root= xn;
      xn = *root - (*root * *root - n) / (2 * *root);
      niter++;
  } while (fabs(xn - *root) > 1e-6);

  return niter;
}
```

```
int square_root_newton2(double n, double *root)
{
  double x, xn = 1.0;
  int niter = 0;

  do {
      x = xn;
      xn = x – (x * x - n) / (2 * x);
      niter++;
  } while (fabs(xn - x) > 1e-6);
  *root = x;
  return niter;
}
```

Assume that 'fabs' is implemented as a built-in that maps to a single instruction. Will one version result in faster code than the other when compiled with an optimizing compiler? If so, state which one. In either case, justify your answer!

No – the two versions result in identical code (tested on gcc IA32). The compiler can and will keep *root in a register because there are no intervening assignments that could change the value of *root. The inner loop updates 'niter' and 'xn', which are local variables – the compiler can determine that the value of '*root' is not changed by those assignments, because 'root' cannot possibly point to a local variable when the function is called.

The most common mistake was to say that you should keep the result in a local variable to avoid unnecessary memory accesses. While this is a good rule in general, and probably the preferred way to write this function, it didn't have any performance benefit in this case. Still, I awarded 6/12 for mentioning it.
A second mistake was to say that fabs() could have side-effects that force *root to be reloaded from memory, but the question stated that 'fabs' is a single-instruction built-in.
For full credit, I expected that you said that it is the absence of aliasing concerns that allows the compiler to keep *root in a register throughout this function.