Programming in C

Lecture 9: Tooling

Dr Neel Krishnaswami

Michaelmas Term 2017-2018

Undefined and Unspecified Behaviour

Undefined and Unspecified Behaviour

- We have seen that C is an *unsafe* language

Undefined and Unspecified Behaviour

- We have seen that C is an *unsafe* language
- Programming errors can arbitrarily corrupt runtime data structures. . .

Undefined and Unspecified Behaviour

- We have seen that C is an *unsafe* language
- Programming errors can arbitrarily corrupt runtime data structures. . .
- . . . leading to *undefined behaviour*

Undefined and Unspecified Behaviour

- We have seen that C is an *unsafe* language
- Programming errors can arbitrarily corrupt runtime data structures...
- ...leading to *undefined behaviour*
- Enormous number of possible sources of undefined behavior (See `https://blog.regehr.org/archives/1520`)

Undefined and Unspecified Behaviour

- We have seen that C is an *unsafe* language
- Programming errors can arbitrarily corrupt runtime data structures...
- ...leading to *undefined behaviour*
- Enormous number of possible sources of undefined behavior (See `https://blog.regehr.org/archives/1520`)
- What can we do about it?

Tooling and Instrumentation

- We have seen that C is an *unsafe* language
- Programming errors can arbitrarily corrupt runtime data structures...
- ...leading to *undefined behaviour*
- There is a great deal of *undefined behaviour*

Tooling and Instrumentation

- We have seen that C is an *unsafe* language
- Programming errors can arbitrarily corrupt runtime data structures...
- ...leading to *undefined behaviour*
- There is a great deal of *undefined behaviour*
- Add instrumentation to detect unsafe behaviour!

Tooling and Instrumentation

- We have seen that C is an *unsafe* language
- Programming errors can arbitrarily corrupt runtime data structures...
- ...leading to *undefined behaviour*
- There is a great deal of *undefined behaviour*
- Add instrumentation to detect unsafe behaviour!
- We will look at 3 tools: ASan, UBSan, and Valgrind

ASan: Address Sanitizer

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`
  - Use stack variable after it is out of scope

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`
  - Use stack variable after it is out of scope
  - Double-frees or other invalid frees

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`
  - Use stack variable after it is out of scope
  - Double-frees or other invalid frees
  - Memory leaks

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`
  - Use stack variable after it is out of scope
  - Double-frees or other invalid frees
  - Memory leaks
- AddressSanitizer instruments code to detect these errors

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`
  - Use stack variable after it is out of scope
  - Double-frees or other invalid frees
  - Memory leaks
- AddressSanitizer instruments code to detect these errors
- Need to recompile

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`
  - Use stack variable after it is out of scope
  - Double-frees or other invalid frees
  - Memory leaks
- AddressSanitizer instruments code to detect these errors
- Need to recompile
- Adds runtime overhead

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`
  - Use stack variable after it is out of scope
  - Double-frees or other invalid frees
  - Memory leaks
- AddressSanitizer instruments code to detect these errors
- Need to recompile
- Adds runtime overhead
- Use it while developing

ASan: Address Sanitizer

- One of the leading causes of errors in C is memory corruption:
  - Out-of-bounds array accesses
  - Use pointer after call to `free()`
  - Use stack variable after it is out of scope
  - Double-frees or other invalid frees
  - Memory leaks
- AddressSanitizer instruments code to detect these errors
- Need to recompile
- Adds runtime overhead
- Use it while developing
- Built into gcc and clang!

ASan Example #1

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define N 10
5
6  int main(void) {
7    char s[N]  = "123456789";
8    for (int i = 0; i <= N; i++)
9      printf ("%c", s[i]);
10   printf("\n");
11   return 0;
12 }
```

ASan Example #1

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define N 10
5
6  int main(void) {
7    char s[N]  = "123456789";
8    for (int i = 0; i <= N; i++)
9      printf ("%c", s[i]);
10   printf("\n");
11   return 0;
12 }
```

- Loop bound goes past the end of the array

ASan Example #1

```c
#include <stdlib.h>
#include <stdio.h>

#define N 10

int main(void) {
  char s[N]  = "123456789";
  for (int i = 0; i <= N; i++)
    printf ("%c", s[i]);
  printf("\n");
  return 0;
}
```

- Loop bound goes past the end of the array
- Undefined behaviour!

ASan Example #1

```c
#include <stdlib.h>
#include <stdio.h>

#define N 10

int main(void) {
  char s[N]  = "123456789";
  for (int i = 0; i <= N; i++)
    printf ("%c", s[i]);
  printf("\n");
  return 0;
}
```

- Loop bound goes past the end of the array
- Undefined behaviour!
- Compile with `-fsanitize=address`

ASan Example #2

```c
#include <stdlib.h>

int main(void) {
  int *a =
    malloc(sizeof(int) * 100);
  free(a);
  return a[5]; // DOOM!
}
```

ASan Example #2

```
1  #include <stdlib.h>
2
3  int main(void) {
4    int *a =
5      malloc(sizeof(int) * 100);
6    free(a);
7    return a[5]; // DOOM!
8  }
```

1. array is allocated

ASan Example #2

```
1  #include <stdlib.h>
2
3  int main(void) {
4    int *a =
5      malloc(sizeof(int) * 100);
6    free(a);
7    return a[5]; // DOOM!
8  }
```

1. array is allocated
2. array is freed

ASan Example #2

```
1  #include <stdlib.h>
2
3  int main(void) {
4    int *a =
5      malloc(sizeof(int) * 100);
6    free(a);
7    return a[5]; // DOOM!
8  }
```

1. array is allocated
2. array is freed
3. array is dereferenced!

ASan Example #2

```c
#include <stdlib.h>

int main(void) {
  int *a =
    malloc(sizeof(int) * 100);
  free(a);
  return a[5]; // DOOM!
}
```

ASan Example #2

```
1  #include <stdlib.h>
2
3  int main(void) {
4    int *a =
5      malloc(sizeof(int) * 100);
6    free(a);
7    return a[5]; // DOOM!
8  }
```

1. array is allocated

ASan Example #2

```
1  #include <stdlib.h>
2
3  int main(void) {
4    int *a =
5      malloc(sizeof(int) * 100);
6    free(a);
7    return a[5]; // DOOM!
8  }
```

1. array is allocated
2. array is freed

ASan Example #2

```
1  #include <stdlib.h>
2
3  int main(void) {
4    int *a =
5      malloc(sizeof(int) * 100);
6    free(a);
7    return a[5]; // DOOM!
8  }
```

1. array is allocated
2. array is freed
3. array is dereferenced!

ASan Example #3

```c
#include <stdlib.h>

int main(void) {
  char *s =
    malloc(sizeof(char) * 10);
  free(s);
  free(s);
  printf("%s", s);
  return 0;
}
```

ASan Example #3

```c
#include <stdlib.h>

int main(void) {
  char *s =
    malloc(sizeof(char) * 10);
  free(s);
  free(s);
  printf("%s", s);
  return 0;
}
```

1. array is allocated

ASan Example #3

```c
#include <stdlib.h>

int main(void) {
  char *s =
    malloc(sizeof(char) * 10);
  free(s);
  free(s);
  printf("%s", s);
  return 0;
}
```

1. array is allocated
2. array is freed

ASan Example #3

```
1  #include <stdlib.h>
2
3  int main(void) {
4    char *s =
5      malloc(sizeof(char) * 10);
6    free(s);
7    free(s);
8    printf("%s", s);
9    return 0;
10 }
```

1. array is allocated
2. array is freed
3. array is double-freed

ASan Limitations

ASan Limitations

- Must recompile code

ASan Limitations

- Must recompile code
- Adds considerable runtime overhead

ASan Limitations

- Must recompile code
- Adds considerable runtime overhead
- Does not catch all memory errors

ASan Limitations

- Must recompile code
- Adds considerable runtime overhead
- Does not catch all memory errors
  - NEVER catches *unitialized* memory accesses

ASan Limitations

- Must recompile code
- Adds considerable runtime overhead
- Does not catch all memory errors
  - NEVER catches *unitialized* memory accesses
  - NEVER catches *unitialized* memory accesses

ASan Limitations

- Must recompile code
- Adds considerable runtime overhead
- Does not catch all memory errors
  - NEVER catches *unitialized* memory accesses
  - NEVER catches *unitialized* memory accesses
- Still: a **must-use** tool during development

UBSan: Undefined Behaviour Sanitizer

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow
  - Dereferencing null pointers

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow
  - Dereferencing null pointers
  - Pointer arithmetic overflow

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow
  - Dereferencing null pointers
  - Pointer arithmetic overflow
  - Dynamic arrays whose size is non-positive

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow
  - Dereferencing null pointers
  - Pointer arithmetic overflow
  - Dynamic arrays whose size is non-positive
- Undefined Behaviour Sanitizer (UBSan) instruments code to detect these errors

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow
  - Dereferencing null pointers
  - Pointer arithmetic overflow
  - Dynamic arrays whose size is non-positive
- Undefined Behaviour Sanitizer (UBSan) instruments code to detect these errors
- Need to recompile

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow
  - Dereferencing null pointers
  - Pointer arithmetic overflow
  - Dynamic arrays whose size is non-positive
- Undefined Behaviour Sanitizer (UBSan) instruments code to detect these errors
- Need to recompile
- Adds runtime overhead

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow
  - Dereferencing null pointers
  - Pointer arithmetic overflow
  - Dynamic arrays whose size is non-positive
- Undefined Behaviour Sanitizer (UBSan) instruments code to detect these errors
- Need to recompile
- Adds runtime overhead
- Use it while developing

UBSan: Undefined Behaviour Sanitizer

- There is lots of non-memory-related undefined behaviour in C:
  - Signed integer overflow
  - Dereferencing null pointers
  - Pointer arithmetic overflow
  - Dynamic arrays whose size is non-positive
- Undefined Behaviour Sanitizer (UBSan) instruments code to detect these errors
- Need to recompile
- Adds runtime overhead
- Use it while developing
- Built into gcc and clang!

UBSan Example #1

```
1  #include <limits.h>
2
3  int main(void) {
4    int n = INT_MAX;
5    int m = n + 1;
6    return 0;
7  }
```

UBSan Example #1

```c
#include <limits.h>

int main(void) {
  int n = INT_MAX;
  int m = n + 1;
  return 0;
}
```

1. Signed integer overflow is undefined

UBSan Example #1

```c
#include <limits.h>

int main(void) {
  int n = INT_MAX;
  int m = n + 1;
  return 0;
}
```

1. Signed integer overflow is undefined
2. So value of `m` is undefined

UBSan Example #1

```c
#include <limits.h>

int main(void) {
  int n = INT_MAX;
  int m = n + 1;
  return 0;
}
```

1. Signed integer overflow is undefined
2. So value of `m` is undefined
3. Compile with `-fsanitize=address`

UBSan Example #2

```
1  #include <limits.h>
2
3  int main(void) {
4    int n = 65
5    int m = n / (n - n);
6    return 0;
7  }
```

UBSan Example #2

```
1  #include <limits.h>
2
3  int main(void) {
4    int n = 65
5    int m = n / (n - n);
6    return 0;
7  }
```

1. Division-by-zero is undefined

UBSan Example #2

```
1  #include <limits.h>
2
3  int main(void) {
4    int n = 65
5    int m = n / (n - n);
6    return 0;
7  }
```

1. Division-by-zero is undefined
2. So value of `m` is undefined

UBSan Example #2

```
1  #include <limits.h>
2
3  int main(void) {
4    int n = 65
5    int m = n / (n - n);
6    return 0;
7  }
```

1. Division-by-zero is undefined
2. So value of `m` is undefined
3. Any possible behaviour is legal!

UBSan Example #3

```
1  #include <stdlib.h>
2
3  struct foo {
4    int a, b;
5  };
6
7  int main(void) {
8    struct foo *x = NULL;
9    int m = x->a;
10   return 0;
11 }
```

UBSan Example #3

```
1  #include <stdlib.h>
2
3  struct foo {
4    int a, b;
5  };
6
7  int main(void) {
8    struct foo *x = NULL;
9    int m = x->a;
10   return 0;
11 }
```

1. Accessing a null pointer is undefined

UBSan Example #3

```c
#include <stdlib.h>

struct foo {
  int a, b;
};

int main(void) {
  struct foo *x = NULL;
  int m = x->a;
  return 0;
}
```

1. Accessing a null pointer is undefined
2. So accessing fields of `x` is undefined

UBSan Example #3

```c
#include <stdlib.h>

struct foo {
  int a, b;
};

int main(void) {
  struct foo *x = NULL;
  int m = x->a;
  return 0;
}
```

1. Accessing a null pointer is undefined
2. So accessing fields of `x` is undefined
3. Any possible behaviour is legal!

UBSan Limitations

UBSan Limitations

- Must recompile code

UBSan Limitations

- Must recompile code
- Adds modest runtime overhead

UBSan Limitations

- Must recompile code
- Adds modest runtime overhead
- Does not catch all undefined behaviour

UBSan Limitations

- Must recompile code
- Adds modest runtime overhead
- Does not catch all undefined behaviour
- Still: a **must-use** tool during development

UBSan Limitations

- Must recompile code
- Adds modest runtime overhead
- Does not catch all undefined behaviour
- Still: a **must-use** tool during development
- **Seriously consider** using it in production

Valgrind

Valgrind

- UBSan and ASan require recompiling

Valgrind

- UBSan and ASan require recompiling
- UBSan and ASan don't catch accesses to unitialized memory

Valgrind

- UBSan and ASan require recompiling
- UBSan and ASan don't catch accesses to unitialized memory
- Enter *Valgrind*!

Valgrind

- UBSan and ASan require recompiling
- UBSan and ASan don't catch accesses to unitialized memory
- Enter *Valgrind*!
- Instruments binaries to detect numerous errors

Valgrind Example

```c
#include <stdio.h>

int main(void) {
  char s[10];
  for (int i = 0; i < 10; i++)
    printf("%c", s[i]);
  printf("\n");
  return 0;
}
```

Valgrind Example

```c
#include <stdio.h>

int main(void) {
  char s[10];
  for (int i = 0; i < 10; i++)
    printf("%c", s[i]);
  printf("\n");
  return 0;
}
```

1. Accessing elements of `s` is undefined

Valgrind Example

```c
#include <stdio.h>

int main(void) {
  char s[10];
  for (int i = 0; i < 10; i++)
    printf("%c", s[i]);
  printf("\n");
  return 0;
}
```

1. Accessing elements of s is undefined
2. Program prints unitialized memory

Valgrind Example

```c
#include <stdio.h>

int main(void) {
  char s[10];
  for (int i = 0; i < 10; i++)
    printf("%c", s[i]);
  printf("\n");
  return 0;
}
```

1. Accessing elements of s is undefined
2. Program prints unitialized memory
3. Any possible behaviour is legal!

Valgrind Example

```c
#include <stdio.h>

int main(void) {
  char s[10];
  for (int i = 0; i < 10; i++)
    printf("%c", s[i]);
  printf("\n");
  return 0;
}
```

1. Accessing elements of `s` is undefined
2. Program prints unitialized memory
3. Any possible behaviour is legal!
4. Invoke `valgrind` with binary name

Valgrind Limitations

Valgrind Limitations

- Adds very substantial runtime overhead

Valgrind Limitations

- Adds very substantial runtime overhead
- Not built into GCC/clang (plus or minus?)

Valgrind Limitations

- Adds very substantial runtime overhead
- Not built into GCC/clang (plus or minus?)
- As usual, does not catch all undefined behaviour

Valgrind Limitations

- Adds very substantial runtime overhead
- Not built into GCC/clang (plus or minus?)
- As usual, does not catch all undefined behaviour
- Still: a **must-use** tool during testing

Assessed Exercise

See "Head of Department's Announcement"

- To be completed by noon on Monday 22 January 2018
- Viva examinations 1330-1630 on Thursday 25 January 2018
- Viva examinations 1330-1630 on Friday 26 January 2018
- Download the starter pack from:
  `http://www.cl.cam.ac.uk/Teaching/1718/ProgC++/`
- This should contain eight files:

  `server.c client.c rfc0791.txt rfc0793.txt`
  `message1 message2 message3 message4`

Exercise aims

Demonstrate an ability to:

- Understand (simple) networking code
- Use control flow, functions, structures and pointers
- Use libraries, including reading and writing files
- Understand a specification
- Compile and test code
- Comprehending man pages

Task is split into three parts:

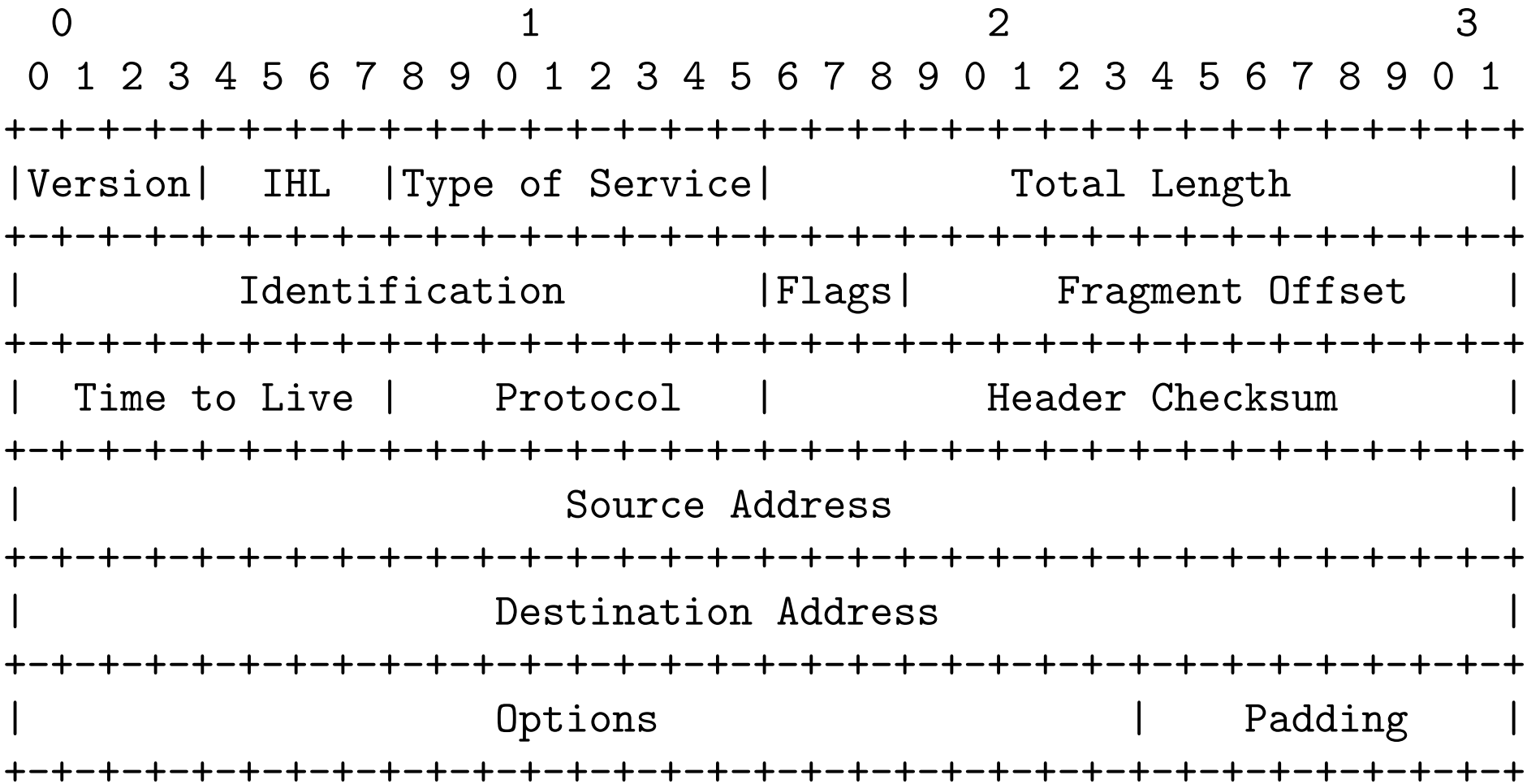
- Comprehension and debugging
- Preliminary analysis
- Completed code and testing

Exercise submission

- Assessment is in the form of a 'tick'
- There will be a short viva; remember to sign up!
- Submission is via email to `c-tick@cl.cam.ac.uk`
- Your submission should include seven files, packed in to a ZIP file called crsid`.zip` and attached to your submission email:

```
answers.txt   client1.c   summary.c   message1.txt
              server1.c   extract.c   message2.jpg
```

Hints: IP header



Hints: IP header (in C)

```c
#include <stdint.h>

struct ip {
  uint8_t hlenver;
  uint8_t tos;
  uint16_t len;
  uint16_t id;
  uint16_t off;
  uint8_t ttl;
  uint8_t p;
  uint16_t sum;
  uint32_t src;
  uint32_t dst;
};

#define IP_HLEN(lenver) (lenver & 0x0f)
#define IP_VER(lenver) (lenver >> 4)
```

Hints: network byte order

- The IP network is big-endian; x86 is little-endian; ARM can be either
- Reading multi-byte values requires possible conversion
- The BSD API specifies:
  - `uint16_t ntohs(uint16_t netshort)`
  - `uint32_t ntohl(uint32_t netlong)`
  - `uint16_t htons(uint16_t hostshort)`
  - `uint32_t htonl(uint32_t hostlong)`

  which encapsulate the notions of host and network and their interconversion (which may be a no-op)