CS 6335 – Language-Based Security

Dr. Kevin Hamlen

A Functional Correctness Proof for *memcpy*

Adem Odza, Avery Arnold, Dagmawet Zemedkun, Omar Khan, Fatema Tuj Johora

Author
François Rozet

December 16, 2025

Contents

1 Introduction 1

2 Technical Approach 3

3 Team Organization 6

4 Evaluation 7

5 Future Work 8

6 Related Work 9

1 Introduction

Our group worked on partially proving the correctness of the *memcpy* function in the MUSL library. The *memcpy* function copies a number of bytes from one section of memory to another.

Figure 1.1. The *memcpy* function header

```
1           #include <string.h>
2           #include <stdint.h>
3           #include <endian.h>
4
5           void *memcpy(void *restrict dest, const void *restrict src, size_t n)
6           {
7               unsigned char *d = dest;
8               const unsigned char *s = src;
9
10          #ifdef __GNUC__
11
12          #if __BYTE_ORDER == __LITTLE_ENDIAN
13          #define LS >>
14          #define RS <<
15          #else
16          #define LS <<
17          #define RS >>
18          #endif
19
20              typedef uint32_t __attribute__((__may_alias__)) u32;
21              uint32_t w, x;
22
23              for (; (uintptr_t)s % 4 && n; n--) *d++ = *s++;
24          ...
25
```

In figure 1.1, we see the header of the *memcpy* function. Three parameters are required: the *dest* pointer where the data will be copied to, the *src* pointer where the data will be copied from, and the *size_t n* which will contain the amount of bytes to copy. Some preprocessor directives are also used to define the bit shift operators depending on the endianess of the

system. Line 23 handles the case where the pointer is misaligned from the word boundary. It copies byte-by-byte until the source pointer is aligned with the word boundary.

Figure 1.2. Copying 16 to 32 bytes

```
25          if ((uintptr_t)d % 4 == 0) {
26              for (; n>=16; s+=16, d+=16, n-=16) {
27                  *(u32 *)(d+0) = *(u32 *)(s+0);
28                  *(u32 *)(d+4) = *(u32 *)(s+4);
29                  *(u32 *)(d+8) = *(u32 *)(s+8);
30                  *(u32 *)(d+12) = *(u32 *)(s+12);
31              }
32              if (n&8) {
33                  *(u32 *)(d+0) = *(u32 *)(s+0);
34                  *(u32 *)(d+4) = *(u32 *)(s+4);
35                  d += 8; s += 8;
36              }
37              if (n&4) {
38                  *(u32 *)(d+0) = *(u32 *)(s+0);
39                  d += 4; s += 4;
40              }
41              if (n&2) {
42                  *d++ = *s++; *d++ = *s++;
43              }
44              if (n&1) {
45                  *d = *s;
46              }
47              return dest;
48          }
49              ...
50
```

The segment of code shown in figure 1.2 handles the copying of bytes when $n >= 16$. 16 byte chunks are copied over until we have less than 16 to copy. At that point, the remaining if statements copy the leftover bytes.

2 Technical Approach

For our proof, we took a lot of inspiration from the proof of the *memset* function. There is one key difference in the behavior of the two function: *memset* sets all modified bytes to the same value (Which is provided by the user) while *memcpy* will set bytes to the corresponding value at the provided source pointer. This similarity allowed for us to create modified versions of the *memset* definitions and apply them to our proof.

The first step in our proof was to create some definitions for certain properties of *memcpy*. We created the *filled* definition to represent the modified memory:

```
Definition filled m dest src len :=
    N.recursion m (fun i m' => m'[(B) dest + i := m (B)[src + i]]) len.
```

This definition states that a memory filled with *len* bytes means that the *len* bytes starting at *dest* and *src* have the same values.

In the invariants section, we defined a few variables:

```
Section Invariants.
    Variable mem : memory    (* initial memory state *).

    Variable dest : N
    Variable src : N
    Variable len : N
```

The first variable *mem* represents the memory state and the rest represent the parameters to the *memcpy* function. The *dest* variable holds the address to the section of memory that will be written to, while *src* holds the address to the section of memory that bytes will be copied from. The *len* variable specifies the amount of bytes to copy.

We also defined a function in the invariant section that returned the intended memory and register state of the function:

```
(* Registers state after copying k bytes *)
Definition memcpy_regs (s : store) k :=
```

```
    s V_MEM64 = filled mem dest src k /\
    s R_X0 = dest /\
    s R_X1 = src /\
    s R_X2 = (len - k).
```

This definition allows us to create an invariant at any point in the function, including during any loop iteration. The value k represents the current progress of the function. If k out of *len* bytes have been filled, then we expect the memory to match the k filled state. We also expect the R_X2 register to have k less than its initial state as the function will subtract from this value to track progress.

Next, we created the invariants so that we could begin our proof.

```
Definition memcpy_invset' (t : trace) : option Prop :=
    match t with
    | (Addr a, s) :: _ =>
        match a with

        (* Entry point *)
        | 0x100000 => Some (memcpy_regs s 0)

        (* Loop 1 (1-byte writes to word boundary) *)
        | 0x100130 => Some (exists mem k, s V_MEM64 = filled mem dest src k)

        (* post-condition: *)
        | 0x100188
        | 0x1000f8
        | 0x100088
        | 0x100064
        | 0x100048
        | 0x100030
        | 0x1000b8 => Some (exists mem, s V_MEM64 = filled mem dest src len)

        | _ => None
        end
    | _ => None
    end.
```

The entry point invariant states that the memory should be unmodified as it should match a $filled\ mem\ dest\ src\ 0$ memory state.

We place the loop invariant where we enter the loop at address $0x100130$. This loop invariant states that for every iteration of the loop, there exists a memory state *mem* and a value k where the current memory state matches $filled\ mem\ dest\ src\ k$. That is, every loop iteration should match the previous one with an extra byte added from the *src* to the *dest* pointer.

The final invariant states that the memory state should equal a fully filled memory state

(*filled mem dest src len*). All *len* bytes at the *src* address should have been copied to the memory at *dest*.

3 Team Organization

Our team consists of 5 people:

- Adem Odza
- Created presentation slides, gave presentation,
- Avery Arnold
- Dagmawet Zemedkun
- Omar Khan
- Fatema Tuj Johora

4 Evaluation

5 Future Work

There are many additions to the proof that could be worked on.

More properties of *memcpy* could also be proved.

Safety properties

6 Related Work

memset