THE UNIVERSITY
OF TEXAS AT DALLAS

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

December 16, 2025

CONTENTS

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.

```
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
```

Figure 1.1. The *memcpy* function header

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.

```
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
```

Figure 1.2. Copying 16 to 32 bytes

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.

In the section of code shown in figure 1.4, we once again copy 16 bytes at a time. Though, this time, we are handling the case where the destination pointer is not aligned to the word boundary. Depending on how many bytes are needed to reach the next word, we enter a different case in the switch statement. Each case then manually copies the bytes needed to reach the next byte boundary. Once that byte boundary is reached, we then copy bytes that are not aligned with the word boundaries. In the for loop, some bits are taken from the current word and the rest are taken from the next word (Depending on which case we are in, this value is different.)

```c
            if (n >= 32) switch ((uintptr_t)d % 4) {
        case 1:
            w = *(u32 *)s;
            *d++ = *s++;
            *d++ = *s++;
            *d++ = *s++;
            n -= 3;
            for (; n>=17; s+=16, d+=16, n-=16) {
                x = *(u32 *)(s+1);
                *(u32 *)(d+0) = (w LS 24) | (x RS 8);
                w = *(u32 *)(s+5);
                *(u32 *)(d+4) = (x LS 24) | (w RS 8);
                x = *(u32 *)(s+9);
                *(u32 *)(d+8) = (w LS 24) | (x RS 8);
                w = *(u32 *)(s+13);
                *(u32 *)(d+12) = (x LS 24) | (w RS 8);
            }
            break;
        case 2:
            w = *(u32 *)s;
            *d++ = *s++;
            *d++ = *s++;
            n -= 2;
            for (; n>=18; s+=16, d+=16, n-=16) {
                x = *(u32 *)(s+2);
                *(u32 *)(d+0) = (w LS 16) | (x RS 16);
                w = *(u32 *)(s+6);
                *(u32 *)(d+4) = (x LS 16) | (w RS 16);
                x = *(u32 *)(s+10);
                *(u32 *)(d+8) = (w LS 16) | (x RS 16);
                w = *(u32 *)(s+14);
                *(u32 *)(d+12) = (x LS 16) | (w RS 16);
            }
            break;
        case 3:
            w = *(u32 *)s;
            *d++ = *s++;
            n -= 1;
            for (; n>=19; s+=16, d+=16, n-=16) {
                x = *(u32 *)(s+3);
                *(u32 *)(d+0) = (w LS 8) | (x RS 24);
                w = *(u32 *)(s+7);
                *(u32 *)(d+4) = (x LS 8) | (w RS 24);
                x = *(u32 *)(s+11);
                *(u32 *)(d+8) = (w LS 8) | (x RS 24);
                w = *(u32 *)(s+15);
                *(u32 *)(d+12) = (x LS 8) | (w RS 24);
            }
            break;
        }
```

3

Figure 1.3. Copying more than 32 bytes

```
100         if (n&16) {
101             *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
102             *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
103             *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
104             *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
105         }
106         if (n&8) {
107             *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
108             *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
109         }
110         if (n&4) {
111             *d++ = *s++; *d++ = *s++; *d++ = *s++; *d++ = *s++;
112         }
113         if (n&2) {
114             *d++ = *s++; *d++ = *s++;
115         }
116         if (n&1) {
117             *d = *s;
118         }
119         return dest;
120
```

Figure 1.4. Leftover bytes

In figure 1.5, the copying of any leftover bytes from the previous loops are handled.

```
25              for (; n; n--) *d++ = *s++;
26              return dest;
27
```

Figure 1.5. Copying for Non GNU Compiler

Figure 1.4 is only executed if the `ifdef __GNUC__` preprocessor check passes. Bytes are simply copied one by one using a for loop before the destination is returned.

2 MOTIVATION

Memcpy is a widely used function that is frequently associated with vulnerabilities such as return-oriented Programming and Buffer overflow attacks. This function is dangerous to use because it does not validate buffer sizes. Attackers can manipulate data in the overflowed areas on the stack to execute malicious code.

A famous case where memcpy played a role was in HeartBleed (CVE-2014-0160). A OpenSSL TLS heartbeat implementation, a user length form was trusted without validation. This length was passed to memcpy without validation leading to attackers reading server memory in each request. The unsafe use of memcpy in the Heartbeat handler caused an out-of-bounds memory read. The impact of this catastrophe led to leakage of passwords and session cookies, ultimately affecting millions of servers worldwide. This meant that authentication was breached and sensitive information was compromised. [1]

Despite being associated with security vulnerabilities, memcpy is fundamental to modern computing systems. This function is used in operating systems, cryptography, and embedded systems. Also they are used in web servers and secure protocols for data movement. The important factor is secure usage of memcpy depends on bounds checking and defense programming because removing memcpy is impractical.

3   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.

3.1   Invariants

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.
```
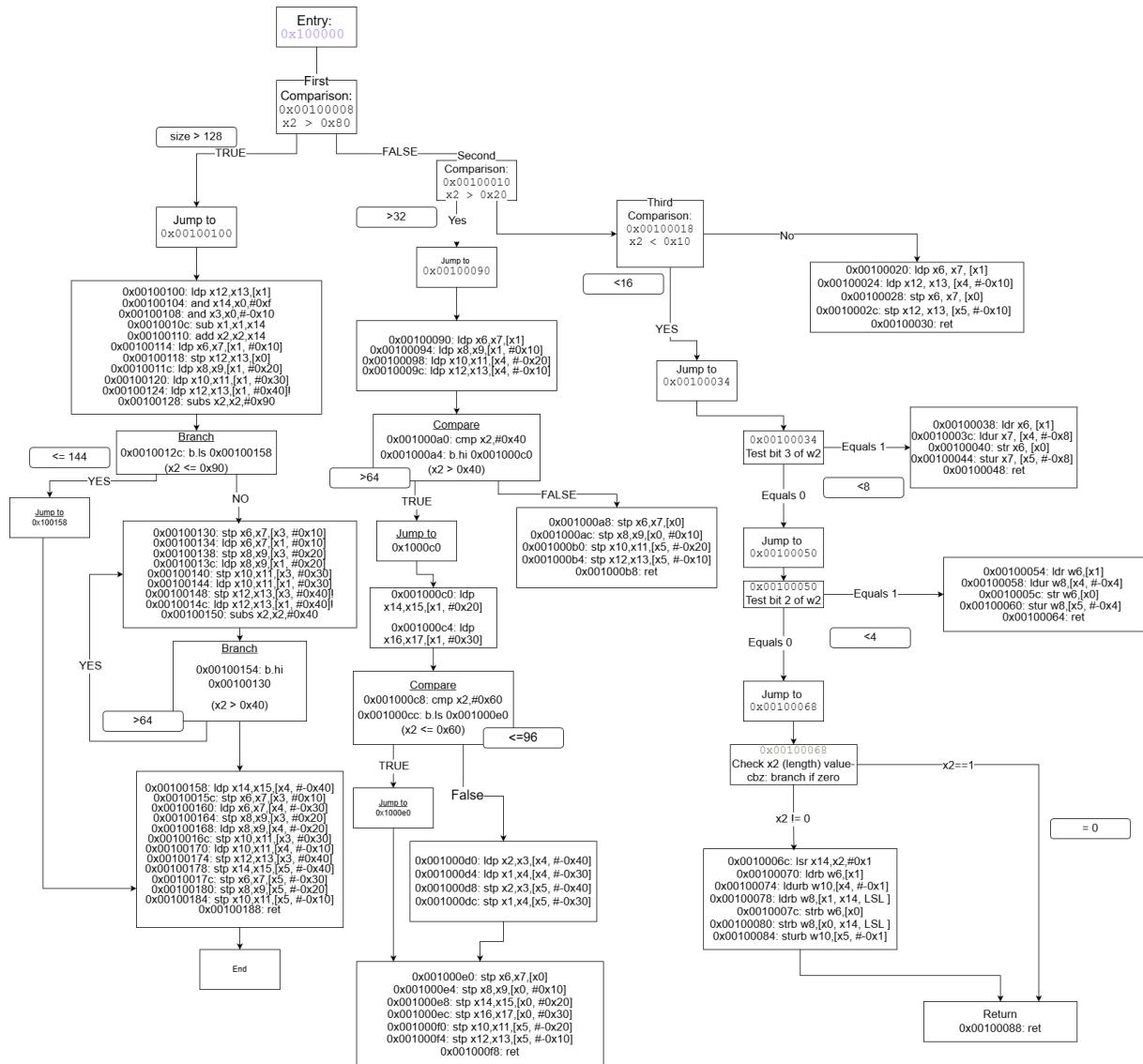
The entry point invariant states that the memory should be unmodified as it should match a

filled mem dest src 0 memory state.

We place the loop invariant where we enter the loop at address 0*x*100130. 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*.

3.2 CONTROL FLOW

Entry:
0x100000

First Comparison:
0x00100008
x2 > 0x80

size > 128 — TRUE

FALSE — Second Comparison:
0x00100010
x2 > 0x20

>32 — Yes

Third Comparison:
0x00100018
x2 < 0x10

No

Jump to
0x00100100

0x00100100: ldp x12,x13,[x1]
0x00100104: and x14,x0,#0xf
0x00100108: and x3,x0,#-0x10
0x0010010c: sub x1,x1,x14
0x00100110: add x2,x2,x14
0x00100114: ldp x6,x7,[x1, #0x10]
0x00100118: stp x12,x13,[x0]
0x0010011c: ldp x8,x9,[x1, #0x20]
0x00100120: ldp x10,x11,[x1, #0x30]
0x00100124: ldp x12,x13,[x1, #0x40]|
0x00100128: subs x2,x2,#0x90

Jump to
0x00100090

0x00100090: ldp x6,x7,[x1]
0x00100094: ldp x8,x9,[x1, #0x10]
0x00100098: ldp x10,x11,[x4, #-0x20]
0x0010009c: ldp x12,x13,[x4, #-0x10]

<16

YES

Jump to
0x00100034

0x00100020: ldp x6, x7, [x1]
0x00100024: ldp x12, x13, [x4, #-0x10]
0x00100028: stp x6, x7, [x0]
0x0010002c: stp x12, x13, [x5, #-0x10]
0x00100030: ret

Branch
0x0010012c: b.ls 0x00100158
(x2 <= 0x90)

<= 144 — YES

Jump to
0x100158

Compare
0x001000a0: cmp x2,#0x40
0x001000a4: b.hi 0x001000c0
(x2 > 0x40)

>64

NO

TRUE

FALSE

0x001000a8: stp x6,x7,[x0]
0x001000ac: stp x8,x9,[x0, #0x10]
0x001000b0: stp x10,x11,[x5, #-0x20]
0x001000b4: stp x12,x13,[x5, #-0x10]
0x001000b8: ret

0x00100034
Test bit 3 of w2

Equals 1

Equals 0

<8

0x00100038: ldr x6, [x1]
0x0010003c: ldur x7, [x4, #-0x8]
0x00100040: str x6, [x0]
0x00100044: stur x7, [x5, #-0x8]
0x00100048: ret

Jump to
0x00100050

0x00100050
Test bit 2 of w2

Equals 1

Equals 0

<4

0x00100054: ldr w6,[x1]
0x00100058: ldur w8,[x4, #-0x4]
0x0010005c: str w6,[x0]
0x00100060: stur w8,[x5, #-0x4]
0x00100064: ret

0x00100130: stp x6,x7,[x3, #0x10]
0x00100134: ldp x6,x7,[x1, #0x10]
0x00100138: stp x8,x9,[x3, #0x20]
0x0010013c: ldp x8,x9,[x1, #0x20]
0x00100140: stp x10,x11,[x3, #0x30]
0x00100144: ldp x10,x11,[x1, #0x30]
0x00100148: stp x12,x13,[x3, #0x40]|
0x0010014c: ldp x12,x13,[x1, #0x40]|
0x00100150: subs x2,x2,#0x40

Branch
0x00100154: b.hi
0x00100130
(x2 > 0x40)

>64

YES

0x001000c0: ldp x14,x15,[x1, #0x20]

0x001000c4: ldp x16,x17,[x1, #0x30]

Compare
0x001000c8: cmp x2,#0x60
0x001000cc: b.ls 0x001000e0
(x2 <= 0x60)

<=96

TRUE

False

Jump to
0x1000e0

Jump to
0x1000c0

Jump to
0x00100068

0x00100068
Check x2 (length) value-
cbz: branch if zero

x2 != 0

x2==1

= 0

0x0010006c: lsr x14,x2,#0x1
0x00100070: ldrb w6,[x1]
0x00100074: ldurb w10,[x4, #-0x1]
0x00100078: ldrb w8,[x1, x14, LSL]
0x0010007c: strb w6,[x0]
0x00100080: strb w8,[x0, x14, LSL]
0x00100084: sturb w10,[x5, #-0x1]

0x00100158: ldp x14,x15,[x4, #-0x40]
0x0010015c: stp x6,x7,[x3, #0x10]
0x00100160: ldp x6,x7,[x4, #-0x30]
0x00100164: stp x8,x9,[x3, #0x20]
0x00100168: ldp x8,x9,[x4, #-0x20]
0x0010016c: stp x10,x11,[x3, #0x30]
0x00100170: ldp x10,x11,[x4, #-0x10]
0x00100174: stp x12,x13,[x3, #0x40]
0x00100178: stp x14,x15,[x5, #-0x40]
0x0010017c: stp x6,x7,[x5, #-0x30]
0x00100180: stp x8,x9,[x5, #-0x20]
0x00100184: stp x10,x11,[x5, #-0x10]
0x00100188: ret

End

0x001000d0: ldp x2,x3,[x4, #-0x40]
0x001000d4: ldp x1,x4,[x4, #-0x30]
0x001000d8: stp x2,x3,[x5, #-0x40]
0x001000dc: stp x1,x4,[x5, #-0x30]

0x001000e0: stp x6,x7,[x0]
0x001000e4: stp x8,x9,[x0, #0x10]
0x001000e8: stp x14,x15,[x0, #0x20]
0x001000ec: stp x16,x17,[x0, #0x30]
0x001000f0: stp x10,x11,[x5, #-0x20]
0x001000f4: stp x12,x13,[x5, #-0x10]
0x001000f8: ret

Return
0x00100088: ret

Below we have included a summarized control flow diagram of our assembly program:

- *n* more than 128 bytes:

9

- Load data from the source at 16 byte (Up to 64) in intervals and store it in the corresponding location at the destination

- In each loop iteration:

  * Copy 64 bytes 2 words at a time

  * Subtract 64 from the remaining value in R_{X0}

  * Once the remaining length is below 64, break from the loop and copy the remaining bytes.

- It then checks $n > 32$:

  - Then $n > 64$

  - $n > 96$

    * Copy the first 32 bytes then fall through to the 64 byte copy.

    * If not, jump to the 64 byte copy and copy the required bytes.

  - $n < 16$

    * If not, copy the double word and return.

  - $n < 8$

    * If not, copy the double word and return.

  - $n < 4$

    * If $n < 4$, check bit values to determine length. Copy the remaining bytes and return.

    * Else, copy the remaining 4 bytes.

3.3 PROOF STRATEGY

Our general proof strategy is as follows:

- Lemmas involving the *filled* definition

  - *filled*0: copying 0 bytes does not change memory

  - *filled_succ*: this lemma proves how to extend a memory copy by one additional byte. k bytes plus an additional byte equals $k + 1$ bytes copied.

  - *filled_add*: this follows similar logic to *filled_succ* but for $k + n$ bytes. This is solved through recursion on n.

  - *filled*1: copying 1 byte in filled is the same as copying 1 byte from *src* to *dst*

  - *filled*2: copying 2 byte in filled is the same as copying 2 byte from *src* to *dst*

  - *filled*3/*filled*3_*pattern*: expands *filled*3 into 3 byte set updates

- *filled*4, *filled*8, *filled*16: helpful lemmas that are specialization of filled_add, for 4 bytes, etc. Simplify reasoning for writes of this size.

- Byte-to-word Conversion: consecutive bytes can combined into word or quad word writes.

  - four_bytes_to_dword: four bytes, one after the other is equivalent to one dword writes.

    * *filled_4_to_dword*: four bytes can be covered by a dword. (write at *dest*)

    * *filled_5_to_dword*: five bytes span two overlapping dwords. (writes at *dest* and *dest* + 1)

    * *filled_6_to_dword*: six bytes span two overlapping dwords. (writes at *dest* and *dest* + 2)

    * *filled_7_to_dword*: seven bytes span two overlapping dwords. (writes at *dest* and *dest* + 3)

    * Other defined lemmas follow this pattern to reason about how our program writes small copies with overlapping dwords.

  - *filled_8_to_qword*: follows similar logic. Eight consecutive bytes is equivalent to 1 qword.

  - *filled_32_to_4qwords* and *filled_64_to_8qwords* follows similar logic.

- Handling overlapping writes: We introduced lemmas to handle unaligned lengths. Copies are handled by this method usually by copying the first bytes (head) and the last bytes (tail). In the case of unaligned lengths, these writes can overlap. The lemmas shows how byte writes of aligned writes plus overlapping tail word writes to prove that the resulting memory is equivalent.

  - *qword_subsumes_bytes_tail*: last bytes of a sequence can be covered by one or more qword writes. qword(8bytes) at dest plus n bytes after it is the same as qword at dest and qword at dest+n.

    * We extend this logic into multiple lemmas reasoning about how this concept holds for different lengths.

  - *filled_overlapmethods*: replace sequences of bytes with the larger writes (either qwords or dword), while accounting for overlapping bytes using the previous lemma about how the tail bytes are covered for the with multiple writes.

    * We use this logic to make a lemma for each possible byte path in our program.

  - *filled_N_toMqwords*: replace many byte-level updates with word-level writes at aligned offsets where *N* is exact multiple of word size of no overlap.

    * We reuse this logic creating multiple lemmas of different *N* qwords consecutive writes.

  - *filled_unaligned_large*: large unaligned byte-level copy equals a small pattern of qwords writes plus bounded overlapping tail qwords.

- *two_qword_writes_to_filled*: fold 16 byte copy into two 8 byte qword writes.

- *filled_lt*4: copies small (1,2,or 3) blocks from start, middle, and end (i.e. copying one-by-one).

We used the respective lemmas to handle each of the length cases mentioned in our control flow graph. At each return point we solve a postcondition (invariant goal) that states the memory at the source and the memory at the destination are the same.

4   TEAM ORGANIZATION

Our team consists of 5 people:

- Adem Odza

  - Initial setup of environment

  - Created invariants

  - Modified *memset* proof statement to fit definition of *memcpy*

  - Modified *memset*'s *filled* definition to fit definition of *memcpy*

  - Created and presented presentation slides

  - Created LaTeX report

- Avery Arnold

  - Mapping of assembly and CFG graph

  - Helped with slides and gave presentation

  - Did the proof and solved length goals

  - Completed Invariant Goals:

    * 8-15 byte copy path

    * 16-32 byte copy path

  - Completed Lemmas:

    * *filled8*

    * *filled_overlap_16to32bytes*

    * *filled_overlap_8bytes*

    * *filled_overlap_8bytes*

  - Admitted Lemmas:

    * *qword_subsumes_bytes_tail_16*

    * *qword_subsumes_bytes_tail*

    * *filled_16_to_2qwords*

- * *filled_8_to_qword*
- * *qwords_subsume_bytes_tail_*32
- Dagmawet Zemedkun
  - Mapping of CFG Graph
  - Tester/Research implementation of connecting k to CPU state
    - * Modification of *memcpy_partial_correctness*
    - * Solved a little over half of the goals
  - Proved 4-7 byte copy path Invariant:
  - Completed lemmas:
    - * *filled*1
    - * *filled*2
    - * *filled_lt*4
  - Solved Code/Proved:
    - * *filled*4
    - * *filled_succ*
    - * Length proof
      - $\cdot$ $0 < len < 4$
      - $\cdot$ $4 <= len < 8$
  - Edited/Wrote:
    - * Final report
      - $\cdot$ Connecting k to CPU state under Future Works
      - $\cdot$ Evaluations
      - $\cdot$ General proof strategy
    - * filled
- Omar Khan
  - Contributed to creating CFG Graph
  - Mapped out entire binary code in Rocq and documented it in comments
  - Adjusted lemmas to fix unsolvable goals (mentioned in evaluation section)
  - Completed Lemmas:
    - * filled16
    - * *filled*3

- * *filled3_pattern*
- * *filled_4_to_dword*
- * *filled_overlap_4bytes*
- * *filled_overlap_32bytes*
- * *filled_overlap_16to32bytes*
- * *filled_overlap_64bytes_small*
- * *filled_overlap_64bytes*
- * *filled_unaligned_large*
- – Admitted Lemmas:
  - * *filled_5_to_dword*
  - * *filled_6_to_dword*
  - * *filled_7_to_dword*
  - * *filled_8_to_qword*
  - * *filled_32_to_4qwords*
  - * *qwords_subsume_bytes_tail_32*
  - * *qword_subsumes_bytes_tail_16*
  - * *filled_64_to_8qwords*
  - * *qwords_subsume_bytes_tail_64_small*
  - * *qwords_subsume_bytes_tail_64*
  - * *unaligned_qword_writes_eq_filled*
- – Completed Invariant Goals:
  - * < 4 byte copy path
  - * 32-64 byte copy path
  - * 64-128 byte copy path
  - * 64-96 byte copy path
  - * 64 byte copy loop path
  - * Large byte copy path
- Fatema Tuj Johora
  - – Formatted and refined presentation slides.
  - – Modified *partial_correctness* theorem (later remodified by team).
  - – Attempted byte-wise and 16–32 byte memory copy proofs.

– Contributed to LaTeX report.

5   Evaluation

At the beginning of the project, we did not map out the binary code given correctly. This led to creating unnecessary invariants on comparisons and loops of multiple sizes making the proof more complicated. To prevent this issue, we should have, at first, defined the correctness theorem and stepped through the code in Rocq to confirm the behavior. The goals stated in Roqc tell us useful information regarding the addresses of the postcondition and entry loop invariants. This helped us create the invariant set and start proving.

Another issue we had as a group was taking too much time focusing on the lemmas. Because we did not have the correctness theorem established, we were blindly defining lemmas that may or may not have been useful. As stated previously, we should have defined the main correctness theorem and stepped through the code until we got to an invariant. This made designing a lemma easier since we had all the information in the invariant goal.

Even after all these struggles, we were able to prove the partial correctness of *memcpy*. Throughout each goal, we had to define lemmas along the way to make the goal easier to prove. Each lemma that was applied in the main proof was proven, however these lemmas depended on other lemmas which were admitted. The lemmas that were admitted were hard to solve due to complexity and lack of understanding of the Picinæ system.

As mentioned, we had to admit some lemma due to their complexity. The lemmas that gave us trouble were related to the hardware representation of memory in Picinæ. We were able to partially unfold their representation using some of the methods included in the Picinæ library. We hit the biggest roadblock when we could not fully unfold *setmem/getmem* to match the representation of both consecutive and overlapping writes on the bit level. Although we had to forgo solving these lemmas in favor of solving the larger proof structure, we are certain that with enough time and a deeper understanding of Picinæ and its abstractions, they are possible to solve.

Along the way, there were some admitted goals relating to length. To solve this issue, we had to adjust the lemmas to make those goals solvable. This was done successfully as we were able to prove that len is less than or equal to 32, 64, 96, and 128 bytes.

In an attempt to solve admitted goals pertaining to impossible length comparisons, we implemented a connection between k and the CPU states. This resulted in altering the partial correctness to specifically state that k bytes were copied and adjust the remaining copyable length. This implementation resulted in altering the flow of the proof but it did not solve its intended purpose of impossible to prove length comparisons. After solving more than half of the goals, it was apparent that either lemmas or assertions were still needed to prove length goals. Hence this implementation was set aside and focus was brought back to the current

solution.

Overall, despite a few false starts and redesigns our group made significant progress on this proof. We gained valuable insight into the challenge of formally verifying programs related to memory. We were able to flush out the main structure of a proof to prove a partial functional correctness of the *memcpy* program. We proved that at each exit point in our program, the bytes at the new destination match the bytes at the source for the length passed to the function. The only missing pieces of our proof were a few small lemmas directly related to the functional representation of memory in the Picinæ system. These lemmas do not affect the overall proof structure or our invariants and require a more fine grained reasoning about the memory model.

6 FUTURE WORK

One thing we would have liked to do if we had more time is prove our few admitted lemmas. In our case these are related directly to the fine grained representation of memory in Picinæ.

6.1 CONNECTING K TO THE CPU STATE

Currently, the solution implemented does not utilize a connection between k and the CPU registers. For future work, it is recommended to implement a connection between k and the CPU. A small tester implementation was attempted but, due to restriction of time, left uncompleted and voided from final submission of project. The core idea of the strategy is to use k as a counter of the bytes that have already been counted. A walk-through of the implementation is provided below.

Invariant Set:

```
| 0x100130 => Some (exists k, k <= len /\ memcpy_regs s k)
```

- Invariant for k bytes copied, where $k <= len$
- Present in the final submission version

Partial Correctness Theorem:

- Original:

  -
    ```
                    forall s dest src len mem t s' x' k
                    (ENTRY: startof t (x', s') = (Addr 0x100000,s))
                    (MDL: models arm8typctx s)
                    (MEM: s V_MEM64 = mem)
                    (R0: s R_X0 = dest)
                    (R1: s R_X1 = src)
                    (R2: s R_X2 = len)
                    (BOUNDS_DEST: dest + len < 2^64)
                    (BOUNDS_SRC : src + len < 2^64)
                    (DIST: (dest + len < src) \/ (src + len < dest))
                    (K_lim: 0 <= k <= len),
    ```

19

```
                    satisfies_all memcpy_lo_memcpy_armv8
                    (memcpy_invset' mem dest src len)
                    program_exit
                    ((x', s')::t).
```

- Updated:

```
                    forall s dest src len mem t s' x' k
                    (ENTRY: startof t (x', s') = (Addr 0x100000,s))
                    (MDL: models arm8typctx s)
                    (MEM: s V_MEM64 = filled mem dest src k)
                    (R0: s R_X0 = dest)
                    (R1: s R_X1 = src)
                    (R2: s R_X2 = len - k)
                    (K_lim: 0 <= k <= len)
                    (BOUNDS_DEST: dest + len < 2^64)
                    (BOUNDS_SRC : src + len < 2^64)
                    (DIST: (dest + len < src) \/ (src + len < dest)),
                    satisfies_all memcpy_lo_memcpy_armv8
                    (memcpy_invset' mem dest src len)
                    program_exit
                    ((x', s')::t).
```

- Specific Lines Changed

    - The *MEM* assumption assumes that the memory follows our filled definition, rather than some arbitrary value.

    - THe *R2* assumption stats that the *R_X2* contains the remaining bytes to copy ($len - k$) rather than the initial target length.

6.2 RESULTS AND DISCUSSION

A slight variation on how to conduct the proof from previous attempts. A negative aspect was that the main issue we were trying to solve: 32 < 32 or other length related impossible goals, were not solved. They remained persistent, contributing to the list of reasons this implementation did not survive into the final submission. Due to these issues, we did not attempt to prove all goals; we stopped after a little after half of the goals were solved. Even though half of the goals were proved, most length related goals were passed with admits. This evaluation is only based on the limited implementation that was attempted; further work and editing of theorems and proofs are needed to make full judgement.

7 RELATED WORK

Verifying the integrity of memory copies is essential for preventing buffer overflow attacks. We took a look at a couple of papers related to this and its impact on the field of research. Specifically, in class we covered a paper on control flow integrity. In this work they cover the importance of defined behavior [3]. In our case, we need to ensure unintended memory behavior did not introduce any security vulnerabilities. Our proof of memcpy strives for this goal, as it verifies byte for bytes memory is copied as intended. We used such principles to map the control flow of the program, and then prove its correctness for each path via well defined behavior.

[2] This paper goes into the function input parameters regarding the destination and source address, and data size. These repeated memory load and store operations depend on the data size and memory alignment. When the source and destination address are properly aligned, larger size memory transfers instructions are used to reduce the number of executed instructions. For small data sizes, memcpy uses byte by byte transfer to ensure correct data copying.

Averill et al. [4] worked on the formal verification of the `memset` function. In this report, they have demonstrated partial verification of `memset`. The verification of `memcpy` is closely related to `memset`, as both functions perform iterative writes over contiguous memory regions and preserve memory safety, boundary constraints. The only difference is that `memcpy` copies values from source memory to destination; whereas `memset` fills a block of memory with a specific byte value. Both functions employ block-wise memory writes and follow largely identical control-flow structures, including separate handling of large and small byte writes through similar branching and looping mechanisms.

Works Cited

[1] I. Ghafoor, I. Jattala, S. Durrani and C. Muhammad Tahir, "Analysis of OpenSSL Heartbleed vulnerability for embedded systems," 17th IEEE International Multi Topic Conference 2014, Karachi, Pakistan, 2014, pp. 314-319, doi: 10.1109/INMIC.2014.7097358.

[2] H. Ying, H. Zhu, D. Wang and C. Hou, "A novel scheme to generate optimal memcpy assembly code," 2013 IEEE Third International Conference on Information Science and Technology (ICIST), Yangzhou, China, 2013, pp. 594-597, doi: 10.1109/ICIST.2013.6747619.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," ACM Transactions on Information and System Security, vol. 13, no. 1, pp. 1–40, Oct. 2009, doi: https://doi.org/10.1145/1609956.1609960.

[4] Averill, C., Kadoi, T., Wank, D., & Harmon, P. (2023). The verification of ARMv7 memset. https://www.charles.systems/writings/The_Verification_of_ARMv7_memset.pdf