The Verification of ARMv7 `memset`

Charles Averill
charles@utdallas.edu
The University of Texas at Dallas
Texas, USA

David Wank
qwe@utdallas.edu
The University of Texas at Dallas
Texas, USA

Takemaru Kadoi
takemaru.kadoi@utdallas.edu
The University of Texas at Dallas
Texas, USA

Payton Harmon
payton.harmon@utdallas.edu
The University of Texas at Dallas
Texas, USA

ABSTRACT

We demonstrate an incomplete, partial verification of the `memset` routine on 32-bit ARMv7 architectures using the Picinæ system. Used to fill a buffer with a provided 8-bit value, `memset` is a core function of the C Runtime Library and therefore should be held to a high level of scrutiny.

ACM Reference Format:
Charles Averill, David Wank, Takemaru Kadoi, and Payton Harmon. 2018. The Verification of ARMv7 `memset`. In *Proceedings of Language-Based Security '23 (LBS).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 MEMSET

On the surface, the generic `memset` algorithm is a relatively trivial routine, even in assembly. However, its optimized MUSL ARMv7 form utilizes a number of loop unrolls, conditional instruction prefixes, and binary arithmetic optimizations that introduce many layers of complexity not only in verification of the function, but in understanding its internal workings at all.

Listing 1: Naïve `memset` Implementation

```
1 void *memset(void *dest, int value, size_t size)
2 {
3     unsigned char *p = dest;
4     while (size-- > 0)
5     {
6         *p++ = value & 255;
7     }
8     return dest;
9 }
```

Listing 1 shows a standard single-loop C implementation of `memset` that sets the contents of `dest` one byte at a time. Figure 1 and Listings 3 and 2 show the control flow, a manual decompilation, and the direct disassembly of the optimized implementation.

The general structure of the code is as follows:

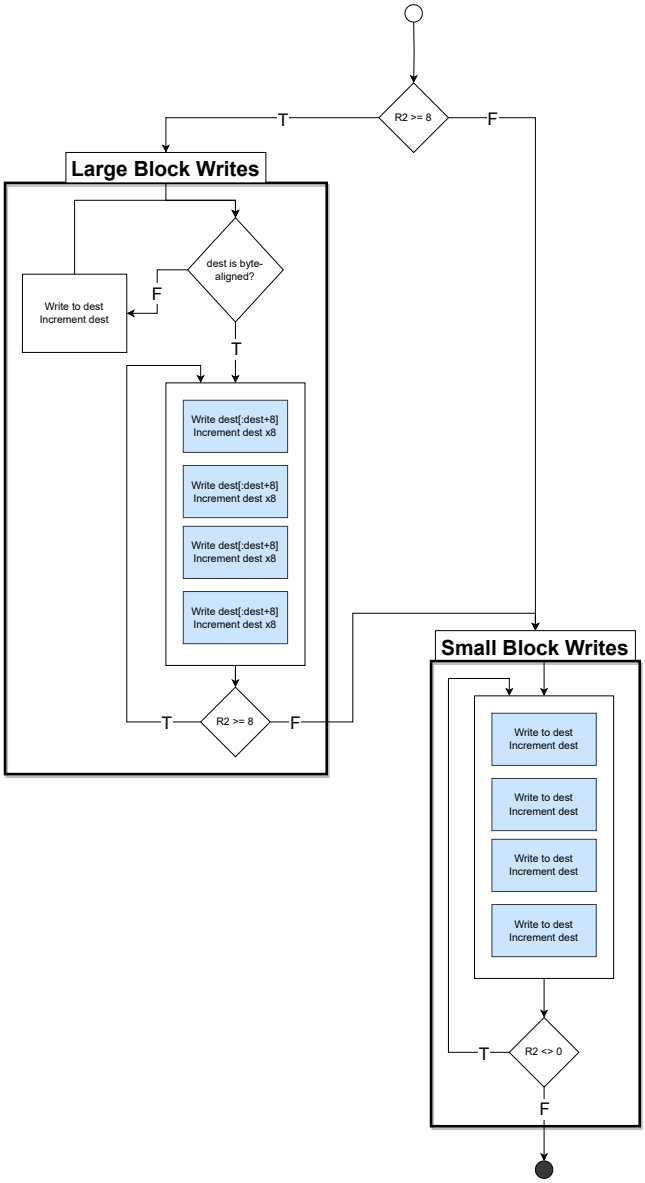

Figure 1: Control Flow of ARMv7 `memset`

LBS, Aug 21–Dec 15, 2023, Richardson, TX

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

1.1 Setup

We will assume the following mapping between registers and their higher-level values:

- `R0` - `dest`, the pointer to write into
- `R1` - `value`, the value to write into `dest`
- `R2` - `size`, the number of positions in memory to write
- `R3` - A copy of `dest`, incremented over time as each write occurs so that an accompanying index variable is not needed. We refer to this as the *moving pointer*
- `R12` - Before Loop 2 executes, `R1` is cast to an 8-bit integer, and then duplicated twice into its upper bits. This value is then copied into `R12`

$$\texttt{R1} = \texttt{R12} = value_{[7:0]}\ value_{[7:0]}\ value_{[7:0]}\ value_{[7:0]}$$

1.2 Loop 1: Byte Alignment

Loop 1 aligns `dest` to a 4-byte boundary (that is, `dest` & 3 = 0). As it's doing so, it writes directly to `dest`, one byte at a time. Alignment is necessary to prepare for the block copies that occur in Loop 2. Block copies are multi-byte move instructions that either require alignment to prevent alignment faults, or prefer alignment to avoid performance penalties, depending on the exact architecture.

1.3 Loop 2: Large Block Writes

Loop 2 performs writes into `dest` in multiple positions in each individual `STM` (Store Multiple) instruction. This loop is further optimized by being partially-unrolled four times. Each unrolled loop instance writes 8 bytes of memory at once, and uses the `HS` flag instruction prefix for conditional execution. The `HS` flag is set only when a subtraction results in a value less than zero. Therefore, by subtracting from `R2` whenever a write occurs, we can track if we have run out of characters to write, and terminate the loop early by skipping over the remaining instructions. In Figure 1, blue blocks signify these conditional executions and a subtraction from `R2`.

1.4 Loop 3: Small Block Writes

Loop 3 follows the same logic as Loop 2, but only performs single-byte writes per unrolled loop.

2 CORRECTNESS

The following correctness specification states that, for all input values of `memset`, every position between `dest` and `dest + size` is equal to the lower 8 bits of `value`.

$$\begin{aligned} &alleq := \forall dest\ value\ size, \\ &\qquad \forall i,\ i < size \implies mem(dest \oplus i) = value\ \%\ 2^8 \end{aligned}$$

A vital aspect to note of the `memset` header is that because it specifies `size`, the number of byte writes to perform, the function cannot loop more than 2^{32} times. This is incredibly helpful, as proving the correctness of any routine dealing with memory will likely have to deal with regions of memory that wrap around the address boundary and intersect with themselves. Occurrences like these essentially never happen in the real world, but that is not a strong enough guarantee to convince Coq.

3 PROOF STRUCTURE

The core of our proof structure, the invariant set, specify points in the program where we expect specific properties to be true. These invariants act as guidelights to coerce our proof to correctness by unifying the state of the program at various points in the execution. Many of our invariants use the following common invariant at their core:

$$\begin{aligned} common\_inv := r1\ \%\ 2^8 = value\ \%\ 2^8 \wedge \\ (\exists k, k \le size \wedge \\ r3 = (dest + k)\ \%\ 2^{32} \wedge (k + r2)\ \%\ 2^{32} = size \wedge alleq\Big) \end{aligned}$$

4 CHALLENGES AND PROGRESS

4.1 Conditional Execution

Many assembly formal verification processes are subject to moments of curiosity, in which semantics of the machine language unexpectedly creates challenges for verifiers. Our example of this was ARM conditional execution.

In the 32-bit ARM ISA, all instructions have a 4-bit condition field which predicates the instruction on the specified condition. If the condition is met, the instruction executes; otherwise, the instruction is effectively a no-op.

In `memset`, the second and third loops of the program are partially unrolled using this mechanism. A store of the desired byte is set predicated on the carry flag, which is then followed by a subtraction from the remaining number of bytes predicated on the same carry flag, which also updates the flag.

This has the effect of attempting, four times in a row, to write more bytes. This is correct because if the carry flag becomes unset by one of the conditional executions (the counter reaches zero), the rest of the attempts in the loop do not execute.

In the context of the Picinæ system, this effectively gets constructed as a control flow graph of the various possibilities of what was executed. We dubbed this the "diamonds problem." Could it be the case that the instructions executed four times, three times, two times, one time, or not at all? As it turns out, the Picinæ system is able to discard impossible cases (one conditional case does not execute, but another later in the loop does) so the problem effectively becomes solving one case, then slightly adjusting it to fit the other slightly different possibilities.

4.2 Accomplishments

This project has successfully completed a number of the initial goals. The common invariant and the correctness specification were written and served as a basis for the rest of the proof. We also completed one of our binary arithmetic helper theorems, proved the first loop, and made significant headway on both the second and their loop.

There are multiple cases that require proving aspects of binary arithmetic necessitating the use of a separate theorem for ease of proving the goals. One of the theorems we proved states:

$$\forall n\ m\ p,\ n \oplus m = n \oplus p \iff m = p$$

We wrote theorems for two other binary arithmetic challenges but have yet to prove them instead opting to admit them and come back to it later.

The byte-alignment loop has been completely proven. Although the first loop is the smallest and simplest of the three, the proof of its correctness used patterns for dealing with memory-modifying code that we've reused in the in-progress proofs for the remaining loops.

The large-block and small-block loops have been partially proven.

4.3 Next Steps

The project ran into a few roadblocks that prevented it from being completed, notably helper theorems that prove aspects of binary arithmetic, as well as the second and the third loop proofs.

We used 3 binary arithmetic theorems, only one of them proven formally. We believe the admitted proofs are true due to near-exhaustive testing but have not completed the proof yet. These theorems are:

$$\forall n\ m\ p\ q,\ n \le m \implies p \le q \implies n \oplus p \le m \oplus q$$

and

$$\forall n\ m,\ n < m \oplus 1 \implies n \le m.$$

When it comes to proving the invariants, there is still a lot of work that needs to be done in both the second and the third loop. Despite having code on both of the proof, each of them contain conditional execution that thoroughly stumped any meaningful progress on the rest of the proof. Loop three is simpler as it writes one byte at a time so the next steps should start there. Once the diamonds challenge is complete as discussed in 4.1 there are two problems that need to be addressed. The first is proving the correctness specification and showing that all written bytes are equal to the value set in the invariant. A subsection of this problems involves proving the the write does not loop back in itself and overlap previous bytes in the array. With the single byte write in the third loop complete the final step after that will be to prove the same challenges in the third loop but that it holds when writing 8 bytes at a time.

4.4 Timeline

The work on `memset` started in Summer 2023, however significant progress began around September. The common invariant was first implemented in early September, leading to a large amount of headway on the byte-alignment loop. Most work done in October was towards proving small binary arithmetic cases and investigating what would be required for the large-block and small-block writes. In the first week of November, we hit our largest milestone: verifying that the first loop of `memset` was indeed correct. Convinced that this was the tipping point, we went forward into proving the remaining loops correct, and ran into the issues with conditional execution that have slowed us down since. The remainder of our time has been spent handling remaining binary arithmetic and conditional execution cases.

A APPENDIX

Listing 2: memset Disassembly

```
undefined __stdcall memset
 undefined         r0:1           <RETURN>
 void *            r0:4           dest
 undefined4        r1:4           value_to_set
 undefined4        r2:4           len
    memset:
    cpy        r3,dest
    cmp        r2,#0x8
    bcc        loop_3
    loop_1:
    tst        r3,#0x3
    strbne     r1,[r3],#0x1
    subne      r2,r2,#0x1
    bne        loop_1
    and        r1,r1,#0xff
    orr        r1,r1,r1, lsl #0x8
    orr        r1,r1,r1, lsl #0x10
    cpy        r12,r1
    loop_2:
    subs       r2,r2,#0x8
    stmiacs    r3!,{r1,r12}
    subcss     r2,r2,#0x8
    stmiacs    r3!,{r1,r12}
    subcss     r2,r2,#0x8
    stmiacs    r3!,{r1,r12}
    subcss     r2,r2,#0x8
    stmiacs    r3!,{r1,r12}
    bcs        loop_2
    and        r2,r2,#0x7
    loop_3  :
    subs       r2,r2,#0x1
    strbcs     r1,[r3],#0x1
    subcss     r2,r2,#0x1
    strbcs     r1,[r3],#0x1
    subcss     r2,r2,#0x1
    strbcs     r1,[r3],#0x1
    subcss     r2,r2,#0x1
    strbcs     r1,[r3],#0x1
    bcs        loop_3
    bx         lr
```

Listing 3: Manually-Decompiled memset

```c
int F_HS = 0;
#define SUB(dest, left, right)
    dest = left - right; F_HS = dest < 0;

void* memset(void* dest, int value, size_t size)
{
    char* r0 = (char*)dest;
    int r1 = value;
    int r12 = 0;
    int r2 = size;
    char* r3 = r0;

    if (r2 >= 8)
    {
        // LOOP 1 : Byte Alignment
        while ((int)r3 & 0b11)
        {
            r3[0] = value;
            r3++;
            SUB(r2, r2, 1);
        }

        // Prepare value
        r1 &= 255;
        r1 |= r1 << 8;
        r1 |= r1 << 16;
        r12 = r1;

        // LOOP 2 : Large Block Writes
        do
        {
            SUB(r2, r2, 8);
            if (!F_HS)
            {
                *((int*)r3) = r1;
                *((int*)(r3 + sizeof(int))) = r12;
                r3 += 2 * sizeof(int);
            }

            if (!F_HS) SUB(r2, r2, 8);
            if (!F_HS)
            {
                *((int*)r3) = r1;
                *((int*)(r3 + sizeof(int))) = r12;
                r3 += 2 * sizeof(int);
            }

            if (!F_HS) SUB(r2, r2, 8);
            if (!F_HS)
            {
                *((int*)r3) = r1;
                *((int*)(r3 + sizeof(int))) = r12;
                r3 += 2 * sizeof(int);
            }

            if (!F_HS) SUB(r2, r2, 8);
            if (!F_HS)
            {
```

```
                *((int*)r3) = r1;
                *((int*)(r3 + sizeof(int))) = ↩
                    r12;
                r3 += 2 * sizeof(int);
            }
        } while (!F_HS);

        // abs(size) if (size > 8)
        r2 &= 7;
    }

    // LOOP 3 : Small Block Writes
    do
    {
        SUB(r2, r2, 1);

        if (!F_HS)
        {
            *r3 = (char)r1;
            r3++;
            SUB(r2, r2, 1);
        }

        if (!F_HS)
        {
            *r3 = (char)r1;
            r3++;
            SUB(r2, r2, 1);
        }

        if (!F_HS)
        {
            *r3 = (char)r1;
            r3++;
            SUB(r2, r2, 1);
        }

        if (!F_HS)
        {
            *r3 = (char)r1;
            r3++;
        }
    } while (!F_HS);

    return r0;
}
```