Architektury systemów komputerowych

Lista zadań nr 6

Na zajęcia 7, 8 i 16 kwietnia 2025

Należy być przygotowanym do wyjaśnienia semantyki każdej instrukcji, która pojawia się w treści zadania. W tym celu posłuż się dokumentacją: x86 and amd64 instruction reference[1]. W szczególności trzeba wiedzieć jak dana instrukcja korzysta z rejestru flag `EFLAGS` tam, gdzie obliczenia zależą od jego wartości.

W trakcie tłumaczenia kodu z asemblera `x86-64` do języka `C` należy trzymać się następujących wytycznych:

- Używaj złożonych wyrażeń minimalizując liczbę zmiennych tymczasowych.
- Nazwy wprowadzonych zmiennych muszą opisywać ich zastosowanie, np. `result` zamiast `rax`.
- Instrukcja `goto` jest zabroniona. Należy używać instrukcji sterowania `if`, `for`, `while` i `switch`.
- Pętle `while` należy przetłumaczyć do pętli `for`, jeśli poprawia to czytelność kodu.

Uwaga! Przedstawienie rozwiązania niestosującego się do powyższych zasad może skutkować negatywnymi konsekwencjami.

Zadanie 1. Poniższy wydruk otrzymano w wyniku deasemblacji rekurencyjnej procedury zadeklarowanej następująco: `long pointless(long n, long *p)`. Zapisz w języku `C` kod odpowiadający tej procedurze. Następnie opisz zawartość jej rekordu aktywacji (ang. *stack frame*). Wskaż rejestry zapisane przez funkcję wołaną (ang. *callee-saved registers*), zmienne lokalne i miejsce, na które będzie wskazywał adres powrotu. Następnie uzasadnij, że wartość rejestru `%rsp` w wierszu 11 jest podzielna przez 16 – zgodnie z [1, 3.2.2]. Zastanów się czemu autorzy ABI zdecydowali się na taką konwencję.

```
1  pointless:
2        pushq   %r14
3        pushq   %rbx
4        pushq   %rax
5        movq    %rsi, %r14
6        movq    %rdi, %rbx
7        testq   %rdi, %rdi
8        jle     .L1
9        leaq    (%rbx,%rbx), %rdi
10       movq    %rsp, %rsi
11       callq   pointless
12       addq    (%rsp), %rax
13       jmp     .L3
14 .L1:  xorl    %eax, %eax
15 .L3:  addq    %rax, %rbx
16       movq    %rbx, (%r14)
17       addq    $8, %rsp
18       popq    %rbx
19       popq    %r14
20       retq
```

Wskazówka: Ta funkcja nie ma konkretnego zastosowania.

Zadanie 2. Poniżej zamieszczono kod rekurencyjnej procedury o sygnaturze `long puzzle2(long *a, long v, uint64_t s, uint64_t e)`. Przetłumacz tę procedurę na język `C`, a następnie jednym zdaniem powiedz co ona robi.

```
1  puzzle2:
2        movq  %rcx, %rax
3        subq  %rdx, %rax
4        shrq  %rax
5        addq  %rdx, %rax
6        cmpq  %rdx, %rcx
7        jb    .L5
8        movq  (%rdi,%rax,8), %r8
9        cmpq  %rsi, %r8
10       je    .L10
11       jg    .L11
12       leaq  1(%rax), %rdx
13       call  puzzle2
14 .L10: ret
15 .L11: leaq  -1(%rax), %rcx
16       call  puzzle2
17       ret
18 .L5:  movq  $-1, %eax
19       ret
```

Wskazówka: Z reguły procedurę `puzzle2` woła się następująco: `i = puzzle2(a, v, 0, n - 1)`.

[1] `http://www.felixcloutier.com/x86/`

Zadanie 3. Zakładamy, że producent procesora nie dostarczył instrukcji skoku (czyli żadnej instrukcji postaci `j*`). Rozważmy procedurę `switch_prob` z poprzedniej listy. Podaj metodę zastąpienia instrukcji `jmpq *0x4006f8(,%rsi,8)` ciągiem innych instrukcji. Nie można używać *kodu samomodyfikującego się* (ang. *self-modifying code*), ani dodatkowych rejestrów.

Napisz kod w języku `C`, który wygeneruje instrukcję *pośredniego wywołania procedury*, na przykład `call *(%rdi,%rsi,8)`, a następnie zaprezentuj go posługując się stroną godbolt[2]. Pokaż, że taką instrukcję też da się zastąpić, gdyby brakowało jej w zestawie instrukcji.

Zadanie 4. Poniżej widnieje kod wzajemnie rekurencyjnych procedur `M` i `F` typu `long (*)(long)`. Programista, który je napisał, nie pamiętał wszystkich zasad *konwencji wołania procedur*. Wskaż co najmniej dwa różne problemy w poniższym kodzie i napraw je! Następnie przetłumacz kod do języka `C`.

```
1  M:   pushq   %rdi
2       testq   %rdi, %rdi
3       je      .L2
4       leaq    -1(%rdi), %rdi
5       call    M
6       movq    %rax, %rdi
7       call    F
8       movq    (%rsp), %rdi
9       subq    %rax, %rdi
10 .L2: movq    %rdi, %rax
11      ret
```

```
12 F:   testq   %rdi, %rdi
13      je      .L3
14      movq    %rdi, %r12
15      leaq    -1(%rdi), %rdi
16      call    F
17      movq    %rax, %rdi
18      call    M
19      subq    %rax, %r12
20      movq    %r12, %rax
21      ret
22 .L3: movl    $1, %eax
23      ret
```

Zadanie 5. Skompiluj poniższy kod źródłowy kompilatorem `gcc` z opcjami `-Og -fomit-frame-pointer -fno-stack-protector` i podejrzyj wygenerowany kod assemblera, ponownie posługując się stroną godbolt. Wskaż w kodzie maszynowym instrukcje realizujące przydział tablicy `p` i narysuj rekord aktywacji procedury. Wyjaśnij co robi instrukcja `leave`.

```
1 long aframe(long n, long idx, long *q) {
2   long i;
3   long *p[n];
4   p[n-1] = &i;
5   for (i = 0; i < n; i++)
6     p[i] = q;
7   return *p[idx];
8 }
```

Zadanie 6. Poniżej widnieje kod procedury o sygnaturze `long puzzle6(void)`. Narysuj rekord aktywacji procedury `puzzle6`, podaj jego rozmiar i składowe. Procedura `readlong`, która wczytuje ze standardowego wejścia liczbę całkowitą, została zdefiniowana w innej jednostce translacji. Jaka jest jej sygnatura? Przetłumacz procedurę `puzzle6` na język `C` i wytłumacz jednym zdaniem co ona robi.

```
1  puzzle6:
2       subq    $24, %rsp
3       movq    %rsp, %rdi
4       call    readlong
5       leaq    8(%rsp), %rdi
6       call    readlong
7       movq    (%rsp), %rax
8       cqto
9       idivq   8(%rsp)
10      xorl    %eax, %eax
11      testq   %rdx, %rdx
12      sete    %al
13      addq    $24, %rsp
14      ret
```

[2] `https://godbolt.org`

Zadanie 7 (bonus). Procedurę ze zmienną liczbą parametrów używającą pliku nagłówkowego `stdarg.h`[3] skompilowano z opcjami `-Og -mno-sse`. Po jej deasemblacji otrzymano następujący wydruk. Przetłumacz procedurę `puzzle7` na język `C` i wytłumacz jednym zdaniem co ona robi. Narysuj rekord aktywacji procedury, a następnie podaj jego rozmiar i składowe. Prezentację zacznij od przedstawienia definicji struktury `va_list` na podstawie [1, 3.5.7].

```
1  puzzle7:
2        movq  %rsi, -40(%rsp)
3        movq  %rdx, -32(%rsp)
4        movq  %rcx, -24(%rsp)
5        movq  %r8, -16(%rsp)
6        movq  %r9, -8(%rsp)
7        movl  $8, -72(%rsp)
8        leaq  8(%rsp), %rax
9        movq  %rax, -64(%rsp)
10       leaq  -48(%rsp), %rax
11       movq  %rax, -56(%rsp)
12       movl  $0, %eax
13       jmp .L2
14 .L3:  movq  -64(%rsp), %rdx
15       leaq  8(%rdx), %rcx
16       movq  %rcx, -64(%rsp)
17 .L4:  addq  (%rdx), %rax
18 .L2:  subq  $1, %rdi
19       js  .L6
20       cmpl  $47, -72(%rsp)
21       ja  .L3
22       movl  -72(%rsp), %edx
23       addq  -56(%rsp), %rdx
24       addl  $8, -72(%rsp)
25       jmp .L4
26 .L6:  ret
```

Literatura

[1] *„System V Application Binary Interface AMD64 Architecture Processor Supplement”*
https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf

[3] https://en.wikipedia.org/wiki/Stdarg.h