LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Arbeitsblatt 4
16.05.2022 - 22.05.2022

T4.1 XOR-Cipher

In dieser Aufgabe werden wir die Funktion

```
void xor_cipher(char* str, int key);
```

in x86-64-Assembler in einer eigenen Datei zu implementieren, welche einen String als Parameter übergeben bekommt und dann jeden einzelnen Buchstaben des Strings mit einer binären XOR-Operation mit dem Parameter `key` verrechnet.

Vorlage: Wir haben für diese Aufgabe eine Vorlage bereitgestellt, die Ihnen das Bearbeiten der Aufgabe erleichtert: `https://gra.caps.in.tum.de/m/xor.tar`

1. Machen Sie sich klar, wie der binäre XOR-Operator funktioniert und lösen Sie damit folgendes Beispiel. Eine ASCII-Tabelle finden Sie unter `man ascii`.

	0	1	0	1	0	1	1	1	ASCII: *'W'*
⊕	***0***	***1***	***1***	***0***	***0***	***0***	***0***	***1***	ASCII: `'a'`
	0	***0***	***1***	***1***	***0***	***1***	***1***	***0***	ASCII: *'6'*

2. Öffnen Sie die Datei `xor.S` und versuchen Sie, jede Zeile zu verstehen.
   - *`.intel_syntax noprefix` **stellt die Syntax auf Intel-Syntax um. Aus historischen Gründen wird standardmäßig die AT&T-Syntax verwendet.***
   - *`.global` `symbol` **exportiert ein Symbol, sodass es auch außerhalb der Assembler-Datei sichtbar ist.***

     Anders als in C, wo alle Funktionen/Variablen standardmäßig extern sichtbar (external linkage) sind, sind Assembly-Label standardmäßig nur intern sichtbar (internal linkage). In C kann man eine Funktion/Variable mit `static` als intern definieren.
   - *`.text` **gibt an, dass der folgende Code in die Section `.text` geschrieben werden soll. Kurzform für `.section .text`***
   - *`your_function:` **ist ein Label, welches den Beginn der Funktion markiert.***
   - *`ret` **springt in die aufgerufene Funktion an die Instruktion nach dem Funktionsaufruf zurück. Die Adresse steht als oberstes Element auf dem Stack.***

3. Passen Sie nun die Datei `xor.S` so an, dass die `xor_cipher` Funktion definiert und exportiert wird. Stellen Sie sicher, dass Ihr Programm mit `make` kompiliert und ausführbar ist.

4. Implementieren Sie unter Einhaltung der Calling Conventions den Rumpf der Funktion `xor_cipher`, die die oben genannte Funktionalität besitzt.

   - Wie werden die Parameter übergeben?[1]

     Die Parameter werden in den Registern `rdi` und `esi` übergeben.

   - Sie werden eine Schleife benötigen, um über die Zeichen des Strings zu iterieren. Welche Abbruchbedingung hat diese?

     Die Schleife muss beim terminierenden NUL-Byte des Strings abbrechen.

   - Was müssen Sie hinsichtlich des Rücksprungs berücksichtigen?

     Alle callee-saved Register müssen denselben Wert haben, wie zuvor; der eigentliche Rücksprung geschieht wie gewohnt mit `ret`.

5. Überprüfen Sie Ihren Programmcode anhand eines selbstgewählten Beispiels. Was kann bei einer ungünstigen Kombination aus Schlüssel und Eingabetext passieren? Wie ließe sich das Problem vermeiden?

   Es kann passieren, dass bereits vor Ende des Strings ein NUL-Byte entsteht, wodurch die Länge bei der Interpretation als C-String verändert wird. Dies ließe sich vermeiden, wenn die Länge des Strings anderweitig gespeichert wird – z.B. durch vorige Berechnung oder durch einen entsprechenden Rückgabewert der Funktion `xor_cipher`.

Vollständige Implementierung der Funktion:

```
1  .intel_syntax noprefix
2  .global xor_cipher
3  .text
4  xor_cipher:
5      cmp byte ptr [rdi], 0
6      je .Lend
7      xor [rdi], sil
8      inc rdi
9      jmp xor_cipher
10 .Lend:
11     ret
```

T4.2 Makefiles

Da ständiges Wiederholen von längeren Kommandos fehleranfällig und schon bei geringfügig größeren Projekten sehr aufwändig ist, werden diese Kommandos meist

[1] Schauen Sie in Aufgabe S4.1, dem Video zur System V ABI, oder fragen Sie Ihren Tutor.

von Build-Systemen automatisch generiert und in separate Dateien geschrieben. In dieser Aufgabe werden wir hierzu die Struktur von *GNU/Unix Makefiles* betrachten.

Vorlage: Wir verwenden hierzu die Materialien der vorigen Aufgabe XOR-Cipher (`https://gra.caps.in.tum.de/m/xor.tar`). Falls Sie die Aufgabe nicht bearbeitet haben, sollten Sie in der Datei `xor.S` den Funktionsnamen auf `xor_cipher` ändern.

1. Öffnen Sie das enthaltene Makefile. Betrachten Sie zunächst folgenden Ausschnitt:

```
main: main.c xor.S
    $(CC) $(CFLAGS) -o $@ $^
```

   Wie interpretieren Sie diese beiden Zeilen? Welche Datei stellt hierbei die zu bauende Zeildatei dar und was sind die zugehörigen Quelldateien?

   Es handelt sich hierbei um eine* Rule*, die immer aus drei Teilen besteht:

```
target: prerequisite1 prerequisite2 ...
    recipe1
    recipe2
```

   Hierbei wird die Datei `target` aus den Dateien `prerequisite1` und `prerequisite2` gebaut, indem die Shell-Befehle `recipe1` und `recipe2` ausgeführt werden. Eine `prerequisite` kann eine Quelldatei sein, es kann aber auch ein anderes* target *sein, welches bei Bedarf ebenfalls gebaut werden würde.* make *überprüft hierbei nicht, dass die Shell-Befehle tatsächlich die Datei `target` bauen – das muss der Autor sicherstellen.

   Hinweis: ***Die Shell-Befehle* müssen *mit Tabs eingerückt werden, nicht mit Leerzeichen!***

2. Variablen werden in Makefiles offensichtlich mit der Syntax `$(varname)` referenziert. Versuchen Sie herauszufinden, wo die beiden Variablen `CC` und `CFLAGS` definiert werden. Nutzen Sie hierzu auch das *GNU-Make Manual*[2].

3. Versuchen Sie nun mittels des *GNU-Make Manuals*[3] herauszufinden, wodurch die Variablen `$@` und `$^` ersetzt werden.

   - ***`$@` ist der Name des* target*s***
   - ***`$^` sind die Namen aller* prerequisites**
   - ***`$<` ist der Name der ersten* prerequisite**

4. Beim Aufführen des Befehls `make` werden als Argumente die *target*s angegeben, die gebaut werden sollen. Wenn kein *target* spezifiziert wird, wird das erste definierte *target* genommen. Zudem lassen sich auch Variablen definieren und weitere Optionen setzen. Finden Sie heraus, was folgende Aufrufe machen:

[2] `https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html`
[3] `https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html`

- `make`

  Hier wird das erste Target gebaut, in diesem Fall *`all`*.

- `make main`

  Hier wird das Target *`main`* ***gebaut.***

- `make clean all`

  Hier wird zuerst das Target *`clean`* ***"gebaut", dann*** *`all`*.

- `make CFLAGS=O3 -Wall -Wextra"`

  Hier wird das erste Target gebaut, zudem werden die *`CFLAGS`* ***angepasst.***

- `make -j2` (nutzen Sie hierzu auch `man make`)

  Spezifiziert, dass zwei Targets gleichzeitig gebaut werden können, z.B. um Mehrkernrechner besser auszunutzen.

5. Führen Sie nun `make clean` aus und danach zwei Mal `make`. Wie erklären Sie sich, dass lediglich beim ersten Durchlauf tatsächlich etwas kompiliert wurde?

   `make` ***verfolgt Dependencies auf Basis der Modfikationszeit. Ist ein*** **target** ***neuer als alle*** **prerequisite**s***, muss es nicht erneut gebaut werden.***

6. Erzeugen Sie nun mit *touch clean* die Datei *clean* und führen Sie `make clean` aus. Funktioniert alles wie erwartet? Wie können Sie das `Makefile` entsprechend ändern, sodass `clean` und `all` immer ausgeführt werden?

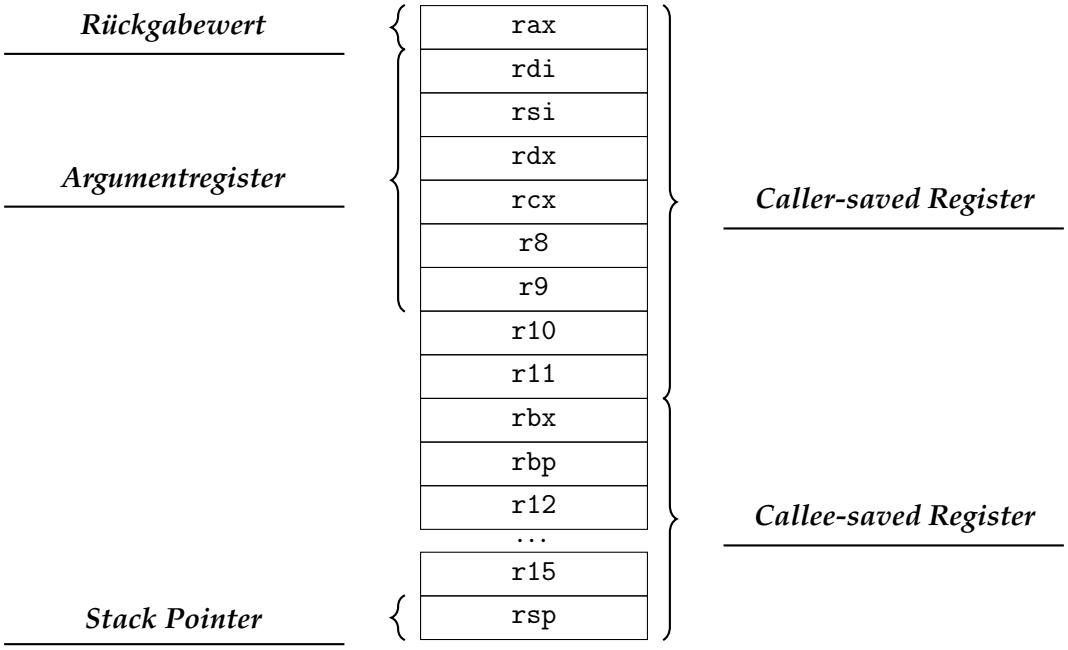
   Hinweis: Nutzen Sie das *GNU-Make Manual*[4].

   Mit *`.PHONY: clean all`* ***– die Beschreibung steht im Manual.***

S4.1 Calling Convention

Wir werden uns in dieser Aufgabe mit der sogenannten *System V Calling Convention* beschäftigen, welche der Compiler beim Aufruf einer Funktion anwendet. Die Calling Convention definiert, wie und wo Parameter übergeben werden, Rückgabewerte erwartet werden, und andere Vorgaben, welche im Zusammenspiel von einer aufgerufenen und einer aufrufenden Funktion zu beachten sind.

[4] `https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html`



1. Tragen Sie in der Grafik die Ihnen bereits bekannte Funktion des Registers `rsp` ein.

2. Betrachten Sie folgendes C-Programm und die zugehörige Disassembly des kompilierten Programms. Können Sie die Register für die ersten beiden Argumente und den Rückgabewert identifizieren?

C-Source	(Dis-)Assembly
`long sub(long a, long b) {`	`mov rax,rdi`
`  return a - b;`	`sub rax,rsi`
`}`	`ret`

Die Operation berechnet $rax = rdi - rsi$ ***– also muss*** `rax` ***den Rückgabewert enthalten,*** `rdi` ***das erste Argument und*** `rsi` ***das zweite Argument.***

3. Können Sie ebenfalls eine Logik erkennen, falls es sehr viele Argumente gibt?

C-Source	(Dis-)Assembly

`long add_sub_many(long a0,`
`  long a1, long a2,`
`  long a3, long a4,`
`  long a5, long a6,`
`  long a7, long a8,`
`  long a9) {`
`  return a0 - a1 + a2 - a3`
`  + a4 - a5 + a6`
`  - a7 + a8 - a9;`
`}`
	`sub rdi,rsi`
	`add rdi,rdx`
	`sub rdi,rcx`
	`add rdi,r8`
	`sub rdi,r9`
	`mov rax,rdi`
	`add rax,qword ptr [rsp+0x8]`
	`sub rax,qword ptr [rsp+0x10]`
	`add rax,qword ptr [rsp+0x18]`
	`sub rax,qword ptr [rsp+0x20]`
	`ret`

Die ersten 6 Integer- und Pointer-Argumente befinden sich in den Registern `rdi`, `rsi`, `rdx`, `rcx`, `r8` und `r9`. Alle weiteren Argumente werden auf den Stack geschrieben. Alle genannten Register, sowie `r10` und `r11` sind temporäre Register und können beliebig verwendet werden – aber auch von aufgerufenen Funktionen überschrieben werden.

4. Wie können Sie sich folgendes Verhalten des Compilers beim Aufruf einer weiteren Funktion erklären? Betrachten Sie insbesondere den Aspekt des Stack-Alignments und bedenken Sie, dass auch durch `call` bereits 8 Byte auf den Stack gelegt werden. Überlegen Sie, wie Sie die Register in folgende zwei Kategorien unterteilen können:

   - Register, die der *aufrufenden Funktion (caller)* gehören und dieser deshalb erwarten kann, dass die Register nach dem Funktionsaufruf denselben Wert haben wie davor.
   - Register, die der *aufgerufenen Funktion (callee)* gehören und deshalb von dieser frei benutzt werden können; für den Aufrufer sind die Registerwerte danach undefiniert.

C-Source	(Dis-)Assembly
	`push rbx` `mov rbx,rdi`

`long sub(long a, long b) {`
`  do_sth(a - b);`
`  return a - b;`
`}`
	`sub rbx,rsi`
	`mov rdi,rbx`
	`call 698 <do_sth>`
	`mov rax,rbx`
	`pop rbx`
	`ret`

Wie bereits erkannt, können einige Register beliebig verwendet und überschrieben werden. Das gilt natürlich auch für andere Funktionen. Die Register `rbx`, `rbp`, `rsp`

und `r12`–`r15` hingegen müssen am Ende der Funktion den gleichen Wert haben wie zu Beginn (Callee-save Register). Dies hat den Vorteil, dass Zwischenergebnisse innerhalb einer Funktion in diesen Registern gespeichert werden können, wenn andere Funktionen aufgerufen werden.

Hier wird also der alte Wert des Registers `rbx` auf dem Stack gespeichert. Dann wird die Berechnung durchgeführt und in das callee-save Register `rbx` geschrieben (was ja nun überschrieben werden kann, da der Wert ja gespeichert wurde). Erst nach dem Aufruf der Funktion wird die Berechnung in das Rückgabewert-Register `rax` geschrieben und anschließend das gespeicherten Register wiederhergestellt.

Hinzu kommt, dass der Stack-Pointer beim Ausführen der Instruktion `call` ein 16-Byte-Alignment haben muss (d.h. die untersten 4 Bits sind 0). Da durch den Aufruf der Funktion selbst mittels `call` die Rücksprungadresse auf den Stack geschrieben wurde ist das Alignment zunächst nicht gegeben und muss vor dem `call` erst hergestellt werden. Hier geschieht dies über die Instruktion `push`.

5. Wie werden Parameter und Rückgabewerte behandelt, die größer als 64 Bit sind?

C-Source	(Dis-)Assembly	
	`mov`	`rax, rdi`
`__int128 inc(__int128 a) {`	`mov`	`rdx, rsi`
`  return a + 1;`	`add`	`rax, 1`
`}`	`adc`	`rdx, 0`
	`ret`	

Ein `__int128` wird wie ein `struct { long low, high; }` behandelt. Da dieses `struct` die Größe von 16 Byte nicht überschreitet, wird es auf zwei Register aufgeteilt. Beim Rückgabewert wird `rdx` als Register für die oberen 64 Bit genommen.

S4.2 Sichere Programmierung

Anders als in Programmiersprachen wie Java gibt es in C (und Assembler) keinen Schutz vor Speicherfehlern. Einer der häufigsten Ursachen von Sicherheitslücken in Programmen sind sog. *Buffer-Overflows*.

1. Was ist ein Buffer-Overflow und was kann dabei passieren? Kann ein Buffer-Overflow von nur einem Byte bereits kritisch sein?

   Buffer Overflows sind Lese- oder Schreibzugriffe auf Speicher jenseits des eigentlichen Buffers (Speicherbereichs). Eine der häufigsten Ursachen für Sicherheitslücken in (kompilierter) Software. Im Worst-Case gibt es Remote Code Execution – ein Angreifer kann die Kontrolle über das System übernehmen. In einigen Fällen genügt sogar ein Overflow von nur einem Byte!

2. Was ist ein *Segmentation Fault*? Wofür ist das Auftreten eines solchen Fehlers ein Indikator? (Was passiert, wenn dies in Ihrer Abgabe der Projektaufgabe passiert?)

   Eine Verletzung des Speicherschutzes von Betriebssystem und Prozessor. Tritt (im Rahmen des Praktikums) i.d.R. bei Null-Pointer-Dereference, Buffer-Overflows oder Verletzungen der Calling-Convention auf.

   Sollte bei Abgaben zu Projektaufgaben unbedingt vermieden werden, Punktabzüge. Insbesondere Randfälle müssen geprüft und abgefangen werden.

3. Sollte man die Funktion `gets` verwenden, um Eingaben von der Konsole einzulesen? Verwenden Sie `man gets`.

   Niemals. Siehe: `man gets` – „Never use this function". Ebenfalls sehr fehleranfällig ist `scanf`. Die empfohlene Alternative ist `fgets`.

S4.3 Kopieren eines Strings

Vergleichen Sie die folgenden Funktionen, welche alle geeignet sind, um einen C-String zu kopieren. Achten Sie insbesondere auf darauf, ob die Funktionen weithin verfügbar sind, was i.d.R. durch Standards sichergestellt ist, ob die Funktionen erkennen lassen, ob der gesamte String kopiert wurde[5], ob das Ergebnis immer ein terminierter String ist, und ob im Sinne der Effizienz nur die notwendigen Bytes geschrieben werden. Ziehen Sie auch die jeweiligen Man-Pages heran.

	`strcpy`	`strncpy`	`stpncpy`	`strlcpy`	`memccpy`
Standardisiert?	***C***	***C***	***POSIX***	***– (BSD)***	***POSIX***
Buffer-Länge spezifizierbar?	—	***Ja***	***Ja***	***Ja***	***Ja***
Rückgabe von String-Ende?	—	—	***Ja***	***Ja***	***Ja***
Schreibt immer NUL-Byte?	***Ja***	—	—	***Ja***	—
Schreibt nur notwendige Bytes?	***Ja***	—	—	***Ja***	***Ja***

Die Funktion `strlcat` ist zwar funktional optimal, aber nicht sehr weit verbreitet und daher nicht standardisiert. Die Funktion `memccpy` ist zur Aufnahme in den C23-Standard vorgesehen.

P4.1 Memccpy [2 Pkt.]

Implementieren Sie die Funktion `memccpy`[6] in C, welche maximal `n` Bytes von `src` nach `dest` kopiert, aber den Kopiervorgang abbricht, *nachdem* ein Byte kopiert wurde, welches

[5] Dadurch lässt sich auch das weitere Anhängen von weiteren Strings erleichtern, da nicht erneut das String-Ende gesucht werden muss.

[6] Siehe auch `man 3 memccpy`.

dem Parameter `c` entspricht. Falls `c` gefunden wurde, gibt die Funktion einen Pointer zum nächsten Byte in `dest` zurück, andernfalls `NULL`.

```
void* memccpy (void* dest, const void* src, int c, size_t n);
```

Referenzlösung:

```
void* memccpy (void* dest, const void* src, int c, size_t n) {
  const unsigned char* s = src;
  for (unsigned char* d = dest; n; n--) {
    *d++ = *s++;
    if (s[-1] == (unsigned char) c)
      return d;
  }
  return NULL;
}
```

P4.2 Map [4 Pkt.]

Implementieren Sie in x86-64 Assembler die Funktion `map`, welche eine Funktion nacheinander auf alle Elemente eines Arrays anwendet und die Werte in dem Array mit den Berechnungsergebnissen aktualisiert.

```
void map(unsigned (*fn)(unsigned), size_t len, unsigned arr[len]);
```

Hinweis: Achten Sie auf *alle* Aspekte der Calling Convention, sowohl in Bezug auf die aufrufende Funktion *als auch* im Bezug auf die Funktion `fn`, die von Ihrer Implementierung aufgerufen wird. Beachten Sie hierbei insbesondere das Stack-Alignment und Caller-saved Register.

Referenzlösung:

```
/* void map(unsigned (*fn)(unsigned), size_t len, unsigned arr[len])
 *
 * Register usage:
 *   rdi - fn
 *   rsi - len
 *   rdx - arr
 *
 *   rbx - fn
 *   r12 - len
 *   r13 - arr
 *   r14 - i
 */
map:
    push rbx
    push r12
    push r13
    push r14

```

```
19      sub rsp, 8
20
21      mov rbx, rdi
22      mov r12, rsi
23      mov r13, rdx
24      xor r14, r14
25
26      jmp .Lloop_check
27
28  .Lloop:
29      mov edi, [r13 + 4*r14]
30      call rbx
31      mov [r13 + 4*r14], eax
32
33      inc r14
34
35  .Lloop_check:
36      cmp r14, r12
37      jb .Lloop
38
39  .Lend:
40      add rsp, 8
41
42      pop r14
43      pop r13
44      pop r12
45      pop rbx
46
47      ret
```

Q4.1 Quiz [4 Pkt.] (siehe Praktikumswebsite)