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

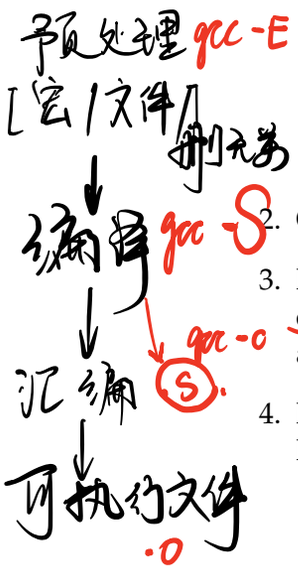
```
void xor_cipher(char* str, int key);
```

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: V
⊕	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.
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]
   - Sie werden eine Schleife benötigen, um über die Zeichen des Strings zu iterieren. Welche Abbruchbedingung hat diese?
   - Was müssen Sie hinsichtlich des Rücksprungs berücksichtigen?
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?

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

T4.2 Makefiles

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

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

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

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.

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:

   - `make`
   - `make main`
   - `make clean all`
   - `make CFLAGS=O3 -Wall -Wextra"`
   - `make -j2` (nutzen Sie hierzu auch `man make`)

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?

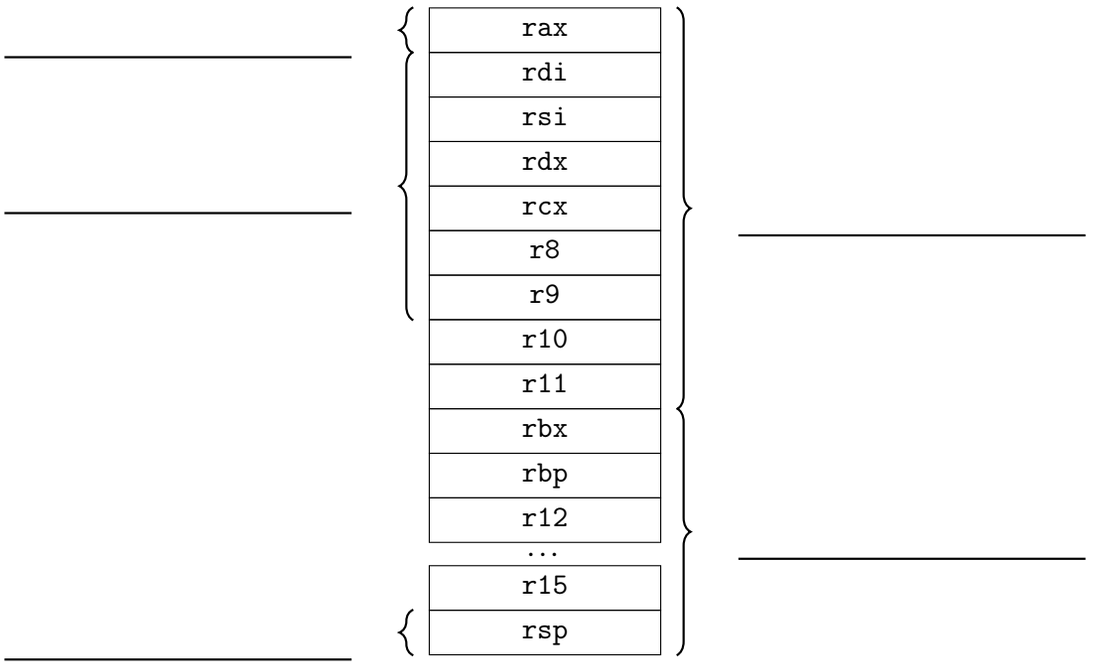
6. Erzeugen Sie nun mit *touch clean* die Datei *clean* und führen Sie `make clean` aus. Funktioniert alles wie erwartet? Wie können Sie das `Makefile` entsprechend ändern, sodass `clean` und `all` immer ausgeführt werden?

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

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

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.



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`

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`

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

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`

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

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?					
Buffer-Länge spezifizierbar?					
Rückgabe von String-Ende?					
Schreibt immer NUL-Byte?					
Schreibt nur notwendige Bytes?					

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 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);
```

[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`.

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.

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