Grundlagenpraktikum Rechnerarchitektur (GRA)

Tutorübung

Moritz Beckel

16.05.2022 16:00 / 20.05.2022 15:00

Hausaufgaben

- Listsum
- Brainfuck
- Quiz

T4.1 XOR-Cipher

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: ____
⊕	_	_	_	_	_	_	_	_	ASCII: 'a'
	_	_	_	_	_	_	_	_	ASCII: ____

T4.1 XOR-Cipher

wget https://gra.caps.in.tum.de/m/xor.tar

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?

T4.1 XOR-Cipher

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?

T4.1 XOR-Cipher

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?

- Problem: durch XOR-Operation entsteht frühzeitig ein Null-Byte
- Länge des Strings wird verändert
- Lösung: Länge anders speichern (bspw. Rückgabe der Funktion)

T4.2 Makefiles

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?

T4.2 Makefiles

Ziel-Datei

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

```
1 target: prerequisite1 prerequisite2 ...
2     recipe1
3     recipe2
```

Benötigte Dateien

Shell-Befehle

TAB (nicht Leertaste)

T4.2 Makefiles

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.

T4.2 Makefiles

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.

CFLAGS: Flags, die dem C-Compiler übergeben werden
CC: Systemspezifischer C-Compiler

$@: Name der Ziel-Datei
$^: Namen aller Voraussetzungen

T4.2 Makefiles

4. Beim Auführen 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?

T4.2 Makefiles

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].

Hausaufgaben

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);
```

Hausaufgaben

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.