Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Hausarbeit für das Proseminar: Python im Hochleistungsrechnen

[CFFI]

Leon Fritz

8fritz@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 7159505
Fachsemester 2

Betreuer: Dr. Michael Kuhn

Inhaltsverzeichnis

1 Einführung

1.1 Einleitung

Für Python-Entwickler ist es von wachsender Wichtigkeit Quelltext zu schreiben welcher nicht nur lesbar, portabel und verständlich, sondern auch performant ist. Python ist, wenn man die Standardimplementierung nutzt, im Vergleich mit Sprachen, welche direkt zu Maschinencode kompiliert werden, relativ langsam und nur schwer erweiterbar. Eine der Hürden dabei ist die jeweilige Python-Implementation, welche man nutzt. Es gibt Implementationen, welche den Python-Quelltext zu Maschinencode kompilieren – z.B. Cython, andere die versuchen die Sprache den Richtlinien nach korrekt zu implementieren – Bsp. CPython, und welche, die versuchen die Sprache mithilfe von Just in Time Kompilierung, z.B. PyPy, zu beschleunigen.

Ein Ansatz, welcher versucht Lesbarkeit, Portabilität, Verständlichkeit, Dynamik, minimale Entwicklerzeit und Performanz zu vereinigen ist CFFI – die Bibliothek, welche im Laufe dieser Hausarbeit vorgestellt wird. CFFI erlaubt es Bestandteile des Programms in C/C++ zu schreiben. Dadurch kann man z.B. Rechnungen mit großen Matrizen [Pyt] beschleunigen. Die Kompatibilität wird dadurch gewährt, dass C-Quelltext auf dem Betriebssystem, auf welchem Python mit CFFI ausgeführt wird, direkt kompiliert wird. Außerdem erlaubt CFFI es auch C/C++ Module, welche bereits auf dem jeweiligen Betriebssystem vorinstalliert sind, aufzurufen [CFFa], weshalb die Portabilität erhalten bleibt. CFFI bildet ein Äquilibrium aus allen Eigenschaften, welche für Python-Code und Entwickler relevant sind. In dieser Hausarbeit wird hauptsächlich auf die Verwendung von CFFI eingegangen. Es wird CPython mit CFFI für alle Code-Beispiele verwendet, auch wenn die Verwendung bei beispielsweise PyPy mit CFFI dieselbe wäre. Anfangs werden die Schnittstellen von CFFI erläutert. Dann wird gezeigt, wie man CFFI verwendet. Am Ende werden die wichtigsten Ergebnisse kurz als Fazit zusammengefasst.

1.2 Motivation

1.2.1 Entwicklerzeit

Einer der Aspekte, weshalb viele überhaupt Python anstelle von beispielsweise C/C++ wählen, ist die Entwicklerzeit. Einen HTTP-Server zu entwickeln dauert in maschinennäheren Sprachen länger als bei Python, da der Entwickler sich genauer mit Speichermanagement, Zeigerarithmetik und mehr auseinandersetzen muss. In Python führt die höhere Abstraktion dazu, dass die Python-Implementation vieles für den Entwickler übernimmt. Dadurch spart der Entwickler Zeit bei der Programmierung und das Unternehmen Geld.

1.2.2 Performance

In vielen Programmen stößt man mit Python an Performancegrenzen. Python eignet sich in seiner Standardimplementierung nicht dazu, schnell und speichereffizient große Berechnungen wie Matrixmultiplikationen durchzuführen, da Python-Quelltext in der Python-Standardimplementierung von einem Interpreter verarbeitet wird. Dadurch gibt es einen großen Overhead zwischen Maschinenebene und Python-Quelltext. Ein Funktionsaufruf in Python muss erst von dem Interpreter in einen „Python-Opcode“ übersetzt werden, welcher dann von einer oder mehreren Routinen als Maschinencode ausgeführt werden kann. Sprachen welche maschinennäher sind, haben deshalb gegenüber der Standardimplementierung von Python einen großen Vorteil. Da ihr Quelltext direkt in den Maschinencode übersetzt wird, entsteht bei der Laufzeit sowie beim Programmstart kein Overhead. Wenn man in einer Sprache wie C++ eine Ganzzahl inkrementiert, dann ist es sehr wahrscheinlich, dass man im Maschinencode auch eine direkte Übersetzung ohne Overhead hat – beim inkrementieren z.B. bei x86 die Instruktion „`inc reg`“. Um dieses Problem zu lösen, gibt es mehrere Ansätze und CFFI ist einer davon.

1.2.3 Vermeiden von Redundanz

C sowie C++ sind seit Jahren etabliert und haben eine große Standardbibliothek sowie eine Vielzahl an Projekten, welche Entwickler im Laufe der Jahre in den Sprachen geschrieben haben. Zwar hat Python auch viele API-Schnittstellen, doch ein Entwickler kann jederzeit etwas finden, das Python standardmäßig nicht hat. Das übliche Vorgehen wäre, selbst Quelltext zu schreiben, welcher genau dieses Verhalten implementiert. Allerdings ist es nicht immer sinnvoll, etwas neu zu erfinden, wenn es dafür bereits geeignete Bibliotheken gibt. Bei kryptographischen Quelltext-Teilen kann dies sogar fatal sein. Deshalb kann man mit CFFI bereits existierende C/C++ Bibliotheken verwenden und dadurch Redundanz vermeiden sowie wichtige Entwicklerzeit sparen (1.2.1).

1.2.4 Hardwarenahe Optimierung

Wenn man nah an der Maschinenebene arbeitet, kann man Optimierungen verwenden, welche bei der Standardimplementierung von Python nicht möglich sind. Ein Beispiel dafür ist der Algorithmus zur schnellen Berechnung der Inverswurzel aus Quake3 [Qua], welcher 1 / sqrt(x) schneller berechnet, als wenn man normale Gleitkommaoperationen verwendet. Der Algorithmus funktionierte nur aufgrund der Maschinennähe und führte dazu, dass das Spiel auf mehr x86 Computern flüssig spielbar war, da die Berechnung immer, wenn ein 3D-Bild gemalt werden sollte, ausgeführt werden musste.

2 Hauptteil

2.1 Schnittstellen von CFFI

2.1.1 ABI – FFI vs API – FFI/Builder

CFFI ist ein vielseitiges Interface und bietet für eine Kommunikation zwischen Python und C/C++ einen ABI-Modus sowie einen API-Modus. Der ABI (Application Binary Interface) Modus arbeitet auf dem ABI-Layer – d.h. er ist plattformabhängig. In dem ABI Modus verwendet man immer vorkompilierte C/C++ Module, beispielsweise die Standard C-Bibliothek oder andere plattformabhängige Bibliotheken (Unix Bibliothek, Windows API, etc.) und ruft ihre Funktionen im Python-Quelltext auf. Dies hat den Nachteil, dass das Python-Programm nur auf den Plattformen funktioniert, auf denen auch der ABI CFFI-Teil des Python-Programms funktioniert. Wenn die Bibliothek nicht auf dem Betriebssystem, auf dem der Python-Code vorhanden ist existiert, dann funktioniert der Quelltext nicht. Somit wird die Plattformunabhängigkeit von Python eingeschränkt. Außerdem muss bei dem ABI-Modus immer jeder Funktionsaufruf zu einer der Bibliotheken durch einen FFI Wrapper laufen. Der Vorteil an dem ABI Modus ist hingegen, dass man sehr einfach Redundanzen (1.2.3) vermeiden kann und dadurch, dass man kein C/C++ schreiben muss Entwicklerzeit sparen (1.2.1). Der API (Application Programming Interface) -Modus hingegen ist etwas aufwendiger, da der Entwickler selbst C schreiben muss. Der API-Modus verwendet den auf dem Betriebssystem vorhandenen C-Kompiler. Man schreibt in seinen Python-Quelltext einen bestimmten Ausschnitt, der in Python langsam wäre (beispielsweise die Matrizenmultiplikation) als C-Quelltext und übergibt den C-Quelltext an CFFI. Dann ändert man sein Python-Programm so, dass das CFFI bei dem ersten Programmstart den Quelltext kompiliert, und wenn dieser bereits kompiliert ist, lediglich ausführt. Der C-Quelltext wird dann zu einer normalen C-Extension mit Python-Kompabilität. Die Extension kann dann mit `from extensionname import ffi, lib` importiert und verwendet werden. Alles, was im C-Quelltext steht wird als nativer Maschinencode ausgeführt. Dadurch hat man dann weiterhin hohe Portabilität und kann hardwarenah optimieren. Außerdem ist die Performance höher, da im Gegensatz zum ABI-Modus nicht jeder Funktionsaufruf durch einen FFI Wrapper laufen muss. Dafür muss der Entwickler allerdings C-Kenntnisse besitzen. und die Entwicklungszeit ist somit höher. Außerdem kann der API-Modus nicht mit C++ umgehen, weshalb man im API Modus nur C schreiben darf. [API]

2.1.2 In-Line vs Out-Of-Line

Der API sowie der ABI Modus haben beide einen „in“ und einen „out“ – line Modus. [CFFb] Im in-line Modus werden bei jeder erneuten Codeausführung C-Funktionsaufrufe neu eingebunden und jeder C-Quelltext neu kompiliert. Stattdessen wird im out-of-line

Modus der C-Quelltext nur einmal kompiliert. Die beste Variante ist der out-of-line API Modus, da dieser laut (2.1.1) in allen Aspekten außer Entwicklerzeit besser ist. Der Entwickler schreibt einen C-Quelltext welcher einmal kompiliert wird und dann als Python-Extension verwendet werden kann. Wenn man im in-line ABI-Modus die C Funktion printf aufruft, dann muss CFFI jedes Mal die Funktionen neu binden und aufrufen. Der out-of-line ABI-Modus ist ein Mix aus dem in-line ABI und den out-of-line API Modus – d.h., wenn man beide verwendet, bezeichnet man dies als out-of-line ABI. Ein Beispiel wäre die Verwendung von selbst geschriebenen C-Code, welcher einmal vorkompiliert wird (out-of-line API-Modus), gekoppelt mit normalen Funktionsaufrufen zur C-Funktion printf (in-line ABI-Modus).

2.1.3 Speichermanagement und Garbagecollection

Python hat anderes Speichermanagement als C. Python gibt Speicher frei, wenn der "Garbagecollector", basierend auf Referenzzählen und ein Zyklen-Erkennungs-Algorithmus, eine Speicherfreigabe anfragt. In C muss der Entwickler selbst dafür sorgen, dass er den Speicher richtig behandelt. Ein C-Entwickler muss, wenn er Speicher auf den Heap alloziert, dafür sorgen, dass er diesen auch wieder freigibt. Dies zu vergessen, kann fatal sein und dazu führen, dass das Programm mit wachsender Laufzeit immer mehr Speicher anfragt, ohne diesen überhaupt zu verwenden. Dadurch entsteht bei der Vermischung von Python und C mithilfe von CFFI ein entscheidendes Problem – wie behandelt man den Speicher bei einem Python-Programm, welches C-Funktionen aufrufen kann? Das Problem besteht nicht, wenn ein Python-Programm mithilfe von CFFI Funktionen aufruft, bei welcher der Heap vor sowie nach Funktionsaufruf bleibt. In dem Fall muss der Entwickler sich um nichts kümmern, außer, dass man die C-Funktion so benutzt wie vorgesehen. Ein Beispiel wäre die C-Funktion clock(), welche parameterlos aufgerufen wird und als Rückgabewert nur eine Ganzzahl liefert. Da nach dem Funktionsaufruf kein Speicher freigegeben werden muss, gibt es dort kein Problem mit der Speicherfreigabe. Wenn man allerdings eine Funktion aufruft, welche Speicher auf dem Heap alloziert lässt nach ihren Funktionsaufruf, beispielsweise `malloc(size)`, muss man diesen in C auch selber löschen. In CFFI wird anstelle von `malloc`, die Python-Funktion `ffi.new` genutzt. Die Funktion `ffi.new("c-datentyp")` liefert eine Referenz, zu einem Objekt, mit einem Datentyp, welcher „cdata“ als Substring hat. Der genaue Python-Datentyp ist vom C-Datentypen abhängig. Bei `ffi.new(„int*“)` ist der Datentyp „`cdata ,int *`“, bei `ffi.new(„int[10]“)` wiederum „`cdata ,int[10]`“. Damit man als Python-Entwickler nicht selbst den Speicher freigeben muss, haben solche Objekte in CFFI einen „Besitzer“ (Bsp. `x = ffi.new(„int*“)` -> Besitzer ist x) [CFFc]. Eine Referenz hat keinen Besitzanspruch auf das Objekt (Bsp. `y = x`). Die einzige Ausnahme ist, wenn man einen Zeiger zu einer C-struct derefenziert. Dadurch erhält man ein Objekt, welches Mitbesitzer ist, da man dadurch ein weiteres Objekt hat, welches von dem adressierten Speicher abhängig ist. Wenn der Besitzer den Speicher nichtmehr referenziert und es kei-

ne Mitbesitzer gibt, dann wird der Speicher automatisch freigegeben. Es ist trotzdem schneller Speichermanagement in C zu betreiben. In Python muss der Garbagecollector immer alle Variablen beobachten. Der Garbagecollector muss selbst herausfinden, wann Speicher freigegeben werden soll. In C steht fest im Quelltext, wann der Speicher gelöscht werden soll (d.h. die Beobachtungsaufgabe entfällt). Außerdem wird C weiterhin kompiliert. Der Aufruf zu `malloc(size)` sowie Speicheroperationen in C erfolgen direkt auf Maschinenebene. Die `ffi.new("c-datentyp")` Funktion hingegen verschachtelt den Aufruf. Sie muss die Zeichenkette mit den Datentypen interpretieren. Um zum selben Ergebnis wie die C-Operation zu kommen, muss jede Heap-Operation immer durch eine Python-Wrapper Funktion und den Python-Interpreter.

2.1.4 Pointer-Arithmetik

In Python kann ein Objekt ähnlich wie in C eine Referenz zu einem anderen Objekt sein. C erlaubt es jedoch, die Referenz zu manipulieren und damit Arithmetik zu betreiben. Man kann in C mithilfe von `&objekt` die Speicher-Addresse von Referenzen erhalten. Um dasselbe in CFFI zu ermöglichen, kann man `ffi.cast(pointer, type)` aufrufen mit der Referenz als ersten Parameter und einen Ganzzahltypen als zweiten. Dadurch erhält man die Adresse der jeweiligen Referenz. Alternativ ist es auch möglich, `ffi.addressof(objekt, offset)` / `ffi.offsetof` zu verwenden. Die Funktionen liefern ebenfalls die Addresse des jeweiligen Objekts allerdings plus den Offset (eine Zahl, welche zur Speicheradresse des Objekts addiert wird) welchen man angegeben hat. Dadurch kann man wie in C einen Zeiger zu einem Objekt erstellen `ffi.addressof(objekt, 0)`, und diese manipulieren – `ffi.addressof(objekt, -50)`. Außerdem kann man in C auch Referenzen dereferenzieren, indem man `*(Referenz)` oder `Referenz[0]` schreibt. Da man in Python standardmäßig nicht `*(Referenz)` schreiben kann aber `Referenz [0]`, hat man bei CFFI `Referenz[0]` als Syntax für die Dereferenzierung verwendet. Wenn man in C eine Referenz hat, kann es sein, dass ihr Wert `NULL` ist. In CFFI gibt es deshalb `ffi.NULL` was wiederum ein Objekt mit den Datentyp `cdata ,type *‘ NULL` liefert.

2.1.5 Typumwandlungen

Eine weitere Funktion, welche es in C und Python gibt, ist die Typumwandlung. Allerdings trennt CFFI die C und die Python Datentypen – jedes Objekt aus C hat den Substring `cdata`. Damit man trotzdem die C-Datentypen umwandeln kann, gibt es die Funktion `ffi.cast(„c datentyp“, wert)`. Die Parameter von `ffi.cast` sind der gewünschte C-Datentyp und einen C-Wert. Eine ungültige Typumwandlung führt zu einem Error. Manche C-Datentypen können in Python-Datentypen umgewandelt werden – ein Integer kann ein Python Integer werden, indem man `int( cinteger )` schreibt.

2.1.6 Umgang mit Klassen und Structs

CFFI erlaubt es nur mit C++ Quelltext zu arbeiten, wenn man den ABI-Modus verwendet d.h. ein vorkompiliertes C++ Modul nutzt. Wenn im ABI-Modus ein C++ Modul eine Klasse zurückgibt, dann wird diese von CFFI erstmal nur als eine normale Speicheradresse behandelt. Da CFFI selbst nicht mit C++ arbeiten kann, hat man nur die Möglichkeit, das Objekt als struct darzustellen. CFFI würde somit jegliche Zugriffsmodifikatoren ignorieren und auf die ganze Klasse zugreifen können. Um eine Klasse zu deklarieren und definieren verwendet man `ffi.cdef`. Um eine Instanz zur Klasse zu erstellen wird `ffi.new` verwendet. Die Eigenschaften des Structs können dann so wie normale Referenzen modifiziert werden. Wenn der Struct beispielsweise eine Zeichenkette des Datentyps `char[10]` namens test enthält, dann kann man mit `instanzname.test = „neuerwert“` diese Eigenschaft modifizieren.

2.1.7 Funktionsaufrufe

Um eine Funktion im ABI-Modus aufzurufen, muss die Funktionen welche man verwenden möchte mit `ffi.cdef("def")` deklarieren (Funktionsprototypen 1:1 aus C kopieren) und dann mithilfe von `dlopen_ret = ffi.dlopen(bibliotheksname)` die Bibliothek angeben, welche dafür zuständig ist, die Funktion zu implementieren. Daraufhin sucht CFFI die Definition der Funktion im Export-Verzeichnis der jeweiligen Bibliothek. Mithilfe von `dlopen_ret.funktionsname(parameter)` kann man sie dann aufrufen. Falls eine deklarierte Funktion nicht gefunden wird, dann wird diese nicht als Schnittstelle angeboten.

2.1.8 Callbacks

Es ist möglich Python Funktionen als Callbacks für C-Funktionen zu verwenden. Man kann beispielsweise eine Python Funktion definieren, welche einen Parameter nimmt und einen Wert zurückgibt. Diese kann dann in eine Callbackfunktion umwandelt werden `ffi.callback(„C-Funktionsprototyp“, Pythonfunktion)`. Die Callbackfunktion kann dann von C-Funktionen aufgerufen werden.

2.1.9 Embedding

Es ist auch möglich, über bloße Funktionsdefinition [CFFd] hinauszugehen und aus Python heraus mithilfe von CFFI eine .so oder .dll zu erstellen, welche von Python als Extension oder von normalen C/C++ Programmen als Bibliothek genutzt werden kann. Dafür ruft man die Funktion `ffi.embedding_api` auf und gibt ihr als Parameter den C-Funktionsprototyp den die Funktion haben soll. Danach setzt man mit `ffi.set_source` den Namen der C-Bibliothek. Dann wird die Funktion implementiert `ffi.embedding_init_code(„Python Quelltext“)`. Der Python-Quelltext ist dabei die Implementation der zuvor deklarierten Funktion. Zuletzt wird der Quelltext kompiliert mithilfe der Funktion `ffi.compile`.

3 CFFI verwenden

3.1 Auf C-Standardbibliothek zugreifen mithilfe des ABI-Modus

Die C-Standardbibliothek bietet eine Vielzahl von Funktionen an. Eine davon ist die C-Funktion „printf“, welche es erlaubt formatierten Text in die Konsole auszugeben. Die Funktion kann man mit CFFI aufrufen, da es eine C-Funktion ist. Der einfachste Weg ist über den ABI-Modus. Zuerst muss die CFFI-Bibliothek importiert und initialisiert werden.

Listing 3.1: CFFI Importieren und initialisieren

```
1 from cffi import FFI
2 ffi = FFI()
```

Jetzt besteht voller Zugriff auf die Schnittstellen von CFFI. Nun wird spezifiziert was für Funktionen gesucht werden sollen, indem die Funktionsdeklaration von `printf` aus der Standard C-Bibliothek kopiert wird.

Listing 3.2: Funktionen deklarieren

```
1 ffi.cdef("int printf(const char *format, ...);")
```

Jetzt wird `ffi.dlopen` verwendet um die Definition von `printf` zu finden.

Listing 3.3: Externes Modul laden

```
1 C = ffi.dlopen("msvcrt.dll")
2 #bei Python3 dlopen("msvcrt.dll"), bei anderen Versionen dlopen(Null)
```

Die Funktion steht nun als Schnittstelle vom Objekt `C` bereit und ist aufrufbar.

Listing 3.4: Funktion aus Ext Modul aufrufen

```
1 C.printf(b"hi there, %s.\n", ffi.new("char[]", b"world"))
```

Das Python-Skript liefert bei Ausführung eine Konsolenausgabe.

Listing 3.5: Programm ausführen

```
1 >hi there, world
```

3.2 Nutzung des API-out-of-line Modus um ein Modul zu erstellen

Der API-Modus ist in jeden Aspekt außer der Entwicklerzeit besser als der ABI-Modus (2.1.1). Deshalb wird jetzt der API-Modus vorgestellt, der zwar schwieriger zu nutzen ist, aber viele Vorteile bietet. Anfangs immer Importierung und Initialisierung.

Listing 3.6: CFFI Importieren und initialisieren

```
1 from cffi import FFI
2 ffi = FFI()
```

Jetzt die Funktionen welche Teil des Moduls werden sollen angeben.

Listing 3.7: Funktionen deklarieren

```
1 ffi.cdef("int printf(const char *format, ...);")
```

Der entscheidende Unterschied ist jetzt, dass nicht in einer fertig kompilierten Bibliothek nach der Definition gesucht wird. Stattdessen definiert der Entwickler im API-Modus die Funktionen selbst. Deshalb muss der Entwickler ffi.set_source aufrufen und die Funktionsdefinition schreiben (oder in diesem Fall, eine vorgefertigte nutzen. Der C-Kompiler liefert für Standardfunktionen selbst die Implementationen).

Listing 3.8: Funktionen definieren

```
1 C = ffi.set_source("meinmodulname",
2     """
3     #include <stdio.h>
4     """
5 )
```

Nun wird das Modul kompiliert.

Listing 3.9: Modul kompilieren

```
1 ffi.compile()
```

Jetzt existiert im Datenverzeichnis eine neue Python-Extension namens `meinmodulname`. Um diese zu nutzen, kann ein Python-Skript geschrieben werden.

Listing 3.10: Modul nutzen

```
1 from meinmodulname import ffi, lib
2 lib.printf(b"wow")
```

Das Python-Skript liefert bei Ausführung eine Konsolenausgabe.

Listing 3.11: Python Programm ausführen

```
>wow
```

3.3 Nutzen eines Nicht-Standardmoduls im in-line ABI-Modus

Im Folgenden wird ein C++ Modul verwendet, welches für diese Hausarbeit erstellt wurde. Auf die Erstellung des Moduls wird nicht eingegangen. Auch wenn ein C Foreign Function Interface der Fokus der Hausarbeit ist, ist die Erstellung des Moduls für die Hausarbeit trivial. Das Modul exportiert 3 native Funktionen - `RPM`, `WPM`, `GetModuleAddress` und eine "Hilfsfunktion `LeyFindPatternStyleIDA`. Das Modul findet intern den Prozess mit den Namen „`hl2.exe`“. Dieser Prozessname wird verwendet von einen Großteil der Spiele welche die Source-Engine nutzen. Das Modul bietet Schnittstellen an, um diesen externen Prozess zu beeinflussen. Die Funktionen `WPM` und `RPM` bieten Lese- und Schreibe-Zugriff auf den Speicher des externen Prozesses. `GetModuleAddress` ermöglicht die Bestimmung der Eingangsaddresse jeglicher Bibliotheken, innerhalb des externen Prozesses. `LeyFindPatternIDAStyle` ermöglicht es eine Aneinanderreihung von Maschinencode-Instruktionen im Quelltext zu finden. Die Funktion `LeyFindPatternIDAStyle` ist in C++ umgesetzt, da der Maschinencode von kompilierten C/C++- Programmen immer groß ist. Somit ähnelt das Finden einer großen Berechnung (1.2.2). Das Ziel ist es, den Prozess so zu modifizieren, dass der Entwicklermodus dauerhaft aktiviert ist und somit jegliche Entwicklerkommandos und Variablen nutzbar sind. In der Spieler-Szene bezeichnet man dies auch als "Cheaten".

Zuerst die Importierung sowie die Initialisierung der CFFI-Bibliothek

Listing 3.12: CFFI Importieren und initialisieren

```
from cffi import FFI
ffi = FFI()
```

Nun folgen die Funktionsdeklarationen, für die Funktionen, welche für das Ziel nützlich sind.

Listing 3.13: Funktionen deklarieren

```
ffi.cdef("""
  int32_t RPM(intptr_t address);
  bool WPM(int address, const char* writethis, int32_t writelen);
  intptr_t GetModuleAddress(const char *module);
```

```
5   intptr_t LeyFindPatternIDAStyle(void* addr, size_t len, const char*↩
        mask);
6 """)
```

Die Definitionen befinden sich in dem vorkompilierten C++ Modul, weshalb es mit der Funktion `ffi.dlopen` geladen werden muss.

Listing 3.14: Funktionen definieren

```
1 #load our module
2 C = ffi.dlopen("manipmem_game.dll")
```

Die Funktionen `GetModuleAddress` sowie `LeyFindPatternIDAStyle` erwarten beide Zeichenketten. Da diese Funktionen für das Ziel notwendig sind, werden die Zeichenketten im Voraus initialisiert. Das Verhalten des Entwicklermodus wird innerhalb der Bibliothek `engine.dll` kontrolliert. Die Zeichenkette `lookforsig` ist eine Aneinanderreihung von x86 Maschinencode-Instruktionen.

Listing 3.15: Zeichenketten initialisieren

```
1 #prepare the strings for the module
2 targetmodule = ffi.new("char[]", b"engine.dll")
3 lookforsig = ffi.new("char[]", b"75 3A 38 05 ? ? ? ? 75 32")
```

Nun wird die Addresse der zu modifizierenden Bibliothek benötigt. Um diese zu bestimmen wird `GetModuleAddress` verwendet.

Listing 3.16: Funktionen aufrufen

```
1 #GetModuleAddress("engine.dll")
2 entryaddr = C.GetModuleAddress(targetmodule)
3 print("Acquired module entry address: " + hex(entryaddr))
```

Eine C/C++ Bibliothek ist groß. Um den zu modifizierenden Code-Abschnitt zu finden wird die Funktion `LeyFindPatternIDAStyle` verwendet. Die Suche führt in diesem Fall zu einen bedingten Sprung. Wenn man in C/C++ ein if-Statement schreibt, vorausgesetzt die Bedingung ist keine Tautologie oder unerfüllbar, wird es in Maschinencode als ein bedingtr Sprung umgesetzt. Im Quelltext des Spiels befindet sich ein if-Statement, welches bestimmt, ob man sich im Entwicklermodus befindet. Dieses if-Statement suchen wir.

Listing 3.17: Gebundene Funktionen aufrufen

```
1
```

```
2 #scan the games memory for our pattern ( sequence of bytes or in our case, sequence of opcodes )
3 targetaddr = C.LeyFindPatternIDAStyle(ffi.cast("void*",entryaddr), 0x30000000, lookforsig);
4 print("Found our pattern at address: " + hex(targetaddr))
```

Jetzt wird das if-Statement überschrieben. Der bedingte Sprung wird mit einen unbedingten ersetzt. Der Entwicklermodus ist somit dauerhaft aktiviert.

Listing 3.18: Gebundene Funktionen aufrufen

```
1 #replace the opcodes with our new ones
2 print("Patching code..")
3 replacement = ffi.new("char[]", b"\xEB");
4 wpmret = C.WPM(targetaddr, replacement, 1); #make it always jmp "if statement is always false"
```

Als letztes wird geprüft ob die Modifikation des externen Prozesses erfolgreich war.

Listing 3.19: Ende des Programms

```
1 if(not wpmret):
2   print("write process memory failed :(")
3
4 #check whether it all worked :)
5 print("Checking whether patch worked")
6 if(C.RPM(targetaddr) == 0xEB):
7   print("Patch worked! you can now enable any cheat convars!")
8 else:
9   print("It did not :(, *(targetaddr) is: " + hex(C.RPM(targetaddr)))
```

Falls es funktioniert hat, sind nun jegliche Entwicklerkommandos, welche in den Zielprozess eingebaut sind, verwendbar. Der Speicher wird nach Programmausführung automatisch von CFFI freigegeben.

Als Test wird nun das Spiel gestartet und ein Spielserver betreten. Danach muss das Python-Programm gestartet werden. Nach Ausführung des Python-Programms ist die Nutzung jeglicher Entwicklerkommandos und Entwicklervariablen möglich. Im folgenden Bild wurde die Entwicklervariable `r_drawothermodels 2` verwendet. Diese erlaubt es, jegliche Gegner und Waffen durch die Wand zu sehen.

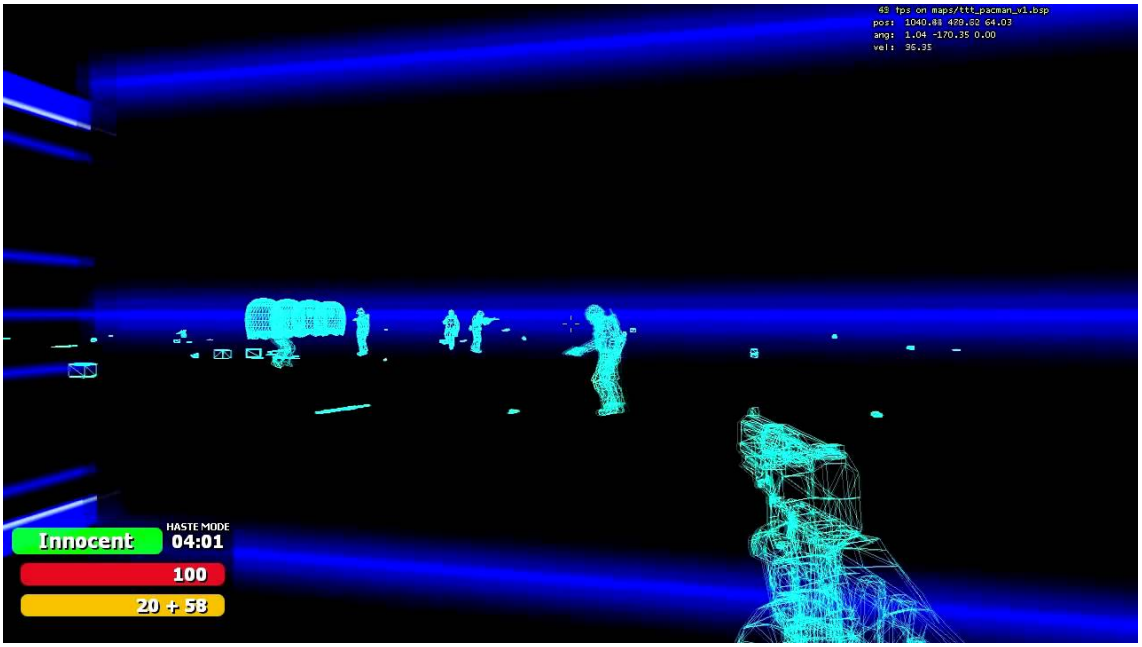

Abbildung 3.1: Dank Entwicklermodus sind nun Variablen nutzbar, welche es erlauben jegliche Gegner und Waffen durch die Wand zu sehen

4 Fazit

Mit CFFI kann selbst ein Python-Anfänger ohne viele C/C++ Vorkenntnisse die Sprache erweitern und langsame Quelltextabschnitte beschleunigen. Die Bibliothek hat viele Anwendungszwecke und Vorteile ihren Alternativen gegenüber. Das Python Ökosystem bietet standardmäßig in seiner Bibliothek die Alternative ctypes an, welche weniger als CFFI kann und in schlechteren, langsameren, aufwendigeren Quelltext resultiert [cffe]. Die Hauptgründe dafür, dass ctypes noch verwendet wird, sind Backwards-Compability sowie der Fakt, dass ctypes fester Bestandteil jeder Python-Implementation ist und ein Umstieg relativ zeitaufwendig. Beim Arbeiten mit CFFI sind klare Vorteile sichtbar, für die Entwicklung sowie für die Laufzeit. Vermutlich wird in der Zukunft irgendwann ctypes als obsolet deklariert und CFFI oder eine Bibliothek, welche ähnlich funktioniert, Teil des Python Standards werden.

Literaturverzeichnis

[API] https://bitbucket.org/cffi/cffi/issues/165/building-c-code-requires-listing-libstdc. Zugriff: 16.08.2019.

[CFFa] https://buildmedia.readthedocs.org/media/pdf/cffi/latest/cffi.pdf. Zugriff: 16.08.2019.

[CFFb] https://cffi.readthedocs.io/en/latest/overview.html?highlight=modes#other-cffi-modes. Zugriff: 16.08.2019.

[CFFc] https://buildmedia.readthedocs.org/media/pdf/cffi/release-0.5/cffi.pdf. Zugriff: 16.08.2019.

[CFFd] https://cffi.readthedocs.io/en/latest/embedding.html. Zugriff: 16.08.2019.

[cffe] http://blog.behnel.de/posts/cython-pybind11-cffi-which-tool-to-choose.html. Zugriff: 16.08.2019.

[Pyt] https://www.benjaminjohnston.com.au/matmul. Zugriff: 16.08.2019.

[Qua] https://web.archive.org/web/20150511044204/http://www.daxia.com/bibis/upload/406Fast_Inverse_Square_Root.pdf. Zugriff: 16.08.2019.