

Obfuscating Lua with a recreated VM

Adam Wang

August 2023

Contents

1	Introduction	1
2	Lua's internals	3
3	Lua Obfuscation	5
4	Filtering the Lua5.1 instructions	6
4.1	Loading constants	6
4.2	Getting from tables	6
4.3	Setting to a table	7
4.4	Functional instructions	7
4.5	Program counter modifications	7
4.6	Other non-reducible instructions	7
5	Creating the Obfuscator	8
5.1	Creating a bytecode interpreter	8
5.2	Methods	9
5.3	Serializing a disassembled chunk	10
5.4	Breaking down the instructions	11
5.5	Generating the VM	12
5.6	Structure of the output	12
6	Conclusion	14
7	Extras	14

1 Introduction

Lua is a scripting language which is lightweight and easily embeddable. Public binaries of Lua5.1 clock in at just under 350KB, and the Lua C API available means that many programs use Lua to act as a scripting language. Examples include Roblox, using the language entirely for game development, Logitech G Hub for macros, and much more. Additionally, it is a fast language even in its

original implementation - other implementations like LuaJIT (Lua with Just-In-Time compilation) are even faster. Comparisons between Lua and Python, a different scripting language, are shown below. The code for both can be found in the Extras section.¹

```
PS C:\Users\adamw\Documents\code\lua-entrails\benchmarking> lua5.1 benchmark.lua
POT      |time taken
6        |0.066
7        |0.656
8        |6.56
9        |64.748
10       |653.564
```

Figure 1: Lua results

```
PS C:\Users\adamw\Documents\code\lua-entrails\benchmarking> python benchmark.py
POT      |time taken
6        |0.23083439999027178
7        |2.350976800022181
8        |29.636328900000008
9        |222.0975481000496
10       |3110.489689800015
```

Figure 2: Python results

There is evidence here to suggest that not only is Lua on average 3-5 times faster than Python when performing the same action, the consistency of the results upon increasing power of tens is much higher with Lua, fluctuating a lot less. Additionally, despite no base implementation of classes, userdata can allocate memory to an undefined object. From Lua, the function `newproxy()` can create userdata with accessible metatables that can implement OOP, or one could just use the extremely powerful table data structure, with the same ability to implement object-oriented programming with metatables. `Lightuserdata` (pointer only) and `userdata` (object) can also be created from the C side, when implementing Lua as an extension. This allows proper objects to be created, and is used in Luau, Roblox's implementation of Lua specifically as a game development language [3].

With this project, I aim to have a deeper understanding of Lua than just its syntax and how one might use it, including PUC-Lua and Luau. Afterwards, I intend to use the information obtained to create an obfuscator for Lua scripts.

¹Even when the Lua script is put through the obfuscator, it is still faster than the Python!

2 Lua's internals

Although Lua is a scripting language, it is compiled to bytecode first (like other language e.g. Python or Java). One advantage quoted about bytecode is that it provides Java the ability to run one compiled file on any machine, however the bytecode which Lua generates is not portable. An example for the simplest compilable block, `do end`, is shown below. Its equivalent listing is also listed underneath. This is generated using `luac`, provided with the binaries of Lua 5.1.4.

```
LuaQ
      @test.lua          €
```

Figure 3: Bytecode output

```
main <do.lua:0,0> (1 instruction, 4 bytes at 0000000000834360)
0+ params, 2 slots, 0 upvalues, 0 locals, 0 constants, 0 functions
      1      [1]      RETURN      0 1
```

Figure 4: Listing provided by Luac

Although the bytecode output may look like a lot of nothing (because it is just mostly whitespace), more can be found out. Using the lua source [1], in `ldump.c` and `lundump.c`, one can make sense of what is happening. `LUAC_HEADERSIZE` is defined as 12 bytes, so we can read the first twelve bytes of the bytecode.

```
\27\76\117\97\81\0\1\4\8\4\8\0
```

Figure 5: 12 bytes extracted from the bytecode, assigned as the header.

Bytes Allocated	Definition and Meaning
4	LUA_SIGNATURE = \033Lua = \27Lua
1	LUAC_VERSION = 0x51 = 81
1	LUAC_FORMAT = 0 (when official PUC)
1	endianness (0=big, 1=little)
1	sizeof(int)
1	sizeof(size_t)
1	sizeof(lua_Number)
1	is lua_Number integral

Figure 6: Information contained in these first 12 bytes

These six flags limit the target machines possible. For example, if a Windows

machine compiled code, the endianness flag would be 1 as Windows is by default nowadays little endian. Linux machines attempting to run this code may run into issues, as the endianness on Linux machines is not guaranteed.

After the header is dumped, the chunks are dumped. Chunks are compiled functions, and the base program is treated as a chunk, therefore one chunk is dumped immediately after the header. Because functions can have functions within them, one can read many chunks within the main chunk itself. This is why the DumpFunction function is indirectly recursive, as each function may require more functions within it to be dumped, depending on what the source code is.

Bytes Allocated	Definition and Meaning
string	Source Name
int	Line defined
int	Last line defined
1	Number of Upvalues
1	Number of Parameters
1	is_vararg
1	Maximum stack size
List	Instructions (DumpCode)
List	Constants (DumpConstants)
List	Debug Data (DumpDebug)

Figure 7: Chunk header data

Strings are encoded with their size first (as a size_t), and then the string itself (followed by a terminator, \0). Empty strings have size 0.

The instructions are the code that is executed. There are 38 opcodes in Lua, all designed to execute a specific instruction enumeration. A normal instruction can be encoded in just a 32-bit integer, where instruction parameters are located at certain offsets when looking at the integer in binary. The three types of instructions are iABC, iABx, iAsBx. As defined in lopcodes.h, six bits are allocated to identify the opcode of the instruction. The A parameter in each is always 8 bits, and the B and C parameters for instructions with type iABC are 9 bits each. For instructions with Bx, where a higher integer is required, the B and C parameters are combined to allocate 18 bits for Bx. For instructions with sBx, which are instructions that require a jump to a different instruction (JMP, FORLOOP, FORPREP), the Bx parameter is a signed integer, allowing for negative numbers as well.

Constants are everything that will (as the name suggests) remain constant. Examples include every single literal in any script. Looking into the code used for benchmarking, every single number that is explicitly written (e.g. 0, 1, 10, 2, 6, 10) and every string (e.g. the format string "%s\t|%s"). Booleans (**true**, **false**) are also included. These are all stored in a constant list in the bytecode, and are loaded by reading one byte defining which byte is to be read. Then they are read like normal (e.g. with strings, as stated before with size first and

then the string itself).

After all important data is deserialized from the bytecode, it is interpreted by the VM. The VM includes a stack (which contains the registers used by each instruction). This imitates a Turing machine, and by executing each instruction as defined in `lopcodes.h`, any script can then be executed.

3 Lua Obfuscation

The verb to obfuscate means to make obscure, unclear, or unintelligible. Standard code practices dictate that code should be readable, without repetition (with the DRY acronym). However, when code in Lua might want to be monetised (examples include `CheckMeIn` on Roblox, a module used in games that the developers have put work into and decided to monetise), measures need to be taken in order to prevent someone from stealing the code. Creativity as well as knowledge is required, as new methods of obfuscation can be simply dreamt up, but a good implementation would be required. Obfuscation exists in other languages as well, intended to slow down an attacker who attempts to maybe reverse engineer a program.

As for methods considered for obfuscation, one simple (and perhaps naive) method of obfuscation is using `loadstring`. Since Lua accepts strings as series of bytes (structured as `\+ number`, e.g. `\27\76\117\97\81\0\1\4\8\4\8\0` as given in the bytecode above), one could simply encode a script into this form and use the `loadstring` function to run it. This type of obfuscation can easily be reversed, however, and one must look deeper in order to create an obfuscation method which gives more control to the developer.

One such method is virtualisation, in the form of VM obfuscation. If the Lua VM can be fully recreated in Lua, the actual code can be provided in a bytecode form (as Lua does), and the VM written in Lua can execute it (referred to now as just the VM). The custom environment created to handle the intermediate bytecode provides a strong layer of protection from a would-be attacker. Lua's compiler, `luac`, is able to strip all debug information from the bytecode output of the original code. This ensures that no attacker can translate the obfuscated output into its original code, with comments, variable names and line number information stripped. However, I wish to look even further. To create such an obfuscator, it must be as difficult as possible to reverse-engineer.

The Turing machine was first invented in 1936 by Alan Turing, used to crack Enigma for World War 2. Although it is simple, Turing proved that a Turing machine can simulate all possible programs [4]. Therefore, a programming language is considered Turing complete if it can simulate a Turing machine. There are 38 different opcodes which correspond to different instructions. If all of these opcodes are used when the VM is recreated, it makes it easier for the attacker to reverse, as patterns can be matched to recognise each opcode and de-obfuscate it into `luac` bytecode, or even to reconstruct the original code. Since we know that Lua is Turing complete and that Turing machines can simulate all possible programs, we know that the Lua instruction set does not increase the function-

ality of the language (rather, it simplifies execution). Each Lua instruction can be remapped to a different instruction set. In this case, it is easier to remap, as the Lua VM is register-based rather than stack-based (even though the data structure which stores the register is called a stack). The Lua 5.1 instruction set can be filtered, and reduced into a different instruction set.

When instructions are replaced with more than 1 instruction, care must be taken to ensure that all jumps are properly handled. If a JMP instruction is meant to jump -3 instructions, but there is an instruction (e.g. LOADNIL) that becomes multiple LOADK instructions, the jump parameter must be increased to compensate for this.

4 Filtering the Lua5.1 instructions

Firstly, all of the instructions need to be grouped up, to see what can be replaced with a combination of other instructions, and what needs to remain. Design decisions need to be made concerning simplicity. An example for this is the constant table - while technically a Turing machine could simulate this, it would be a waste of effort and computer memory to have to run a whole series of instructions just so constants are loaded. Although ease of reverse-engineering needs to be kept in mind, so does pure computational power - each obfuscated script should have a reasonable execution speed. Implementations of each opcode should still be kept though, as in certain edge cases it may not be possible to simply reduce instruction behaviour.

All opcode mnemonics (in capital letters) used here are the same as in the 5.1 source [1]: see `lopcodes.h` for a list.

4.1 Loading constants

As mentioned before, we cannot eliminate the constant table, but we can eliminate less useful instructions from this set. As established in `lopcodes.h`, the instructions which are in this group are MOVE, LOADK, LOADBOOL, LOADNIL, GETUPVAL. Similar to the constant table, the upvalue table will also be kept. However, LOADBOOL and LOADNIL can all be optimised. Both LOADNIL and LOADBOOL have secondary uses - a single LOADNIL instruction can set a range of registers to nil. However, we can replace this with many LOADK instructions. The Bx parameter in LOADK, used for the index in the constant table, can be replaced with a 0. Using metatables, we can modify the behaviour of the constant table to return `nil` if `Constants[0]` is indexed. Similarly, we can add `true` and `false` to the constant table, and replace the LOADBOOL instruction with another LOADK.

4.2 Getting from tables

The next section defined in `lopcodes.h` contains GETGLOBAL and GETTABLE. One possible way would be to cut out GETGLOBAL and have the B parameter

of GETTABLE index a special value, possibly less than or equal to 0. However, considering GETGLOBAL is iABx, it must be carefully checked whether the Bx value is greater than the maximum value allowed for C (511). If it is, this method cannot be used.

Additionally, the SELF instruction prepares for a method call and is exactly a MOVE and a GETTABLE instruction's functionality. This can be easily broken up into two instructions.

4.3 Setting to a table

This section includes SETGLOBAL, SETUPVAL, and SETTABLE. Similarly to getting from tables, we could allocate special indexes for the A parameter to mean the global table, or the upvalue table. Although SETUPVAL is iABC, SETGLOBAL is not, and the same issues could arise when attempting to index the constant table with values above 511.

4.4 Functional instructions

In this collection, not grouped in the source, I have chosen to group CALL, TAILCALL, RETURN, CLOSE, CLOSURE, and SELF. For most of these, the implementation of each instruction is either difficult to redo or difficult to break down. However, TAILCALL is used for one specific return style. As documented (`return R(A) (R(A+1), ... ,R(A+B-1))`), TAILCALL is used just to avoid nesting calls one layer deeper, and therefore it can be replaced with a CALL and a RETURN.

4.5 Program counter modifications

In each definition of an opcode where the program counter is incremented as part of its functionality, a comparison is made and the program counter is incremented as part of a conditional statement. If these were to be broken up, the results of this conditionality test must be able to be passed between instructions, and this requires the use of a separate state variable (modifications to the stack require complicated instruction reworks, which are not suitable at a beginner level). Therefore, to begin with (in order to keep the virtual machine stateless), each of these variables cannot be broken down into a separate JMP instruction. These instructions are LOADBOOL, EQ, LT, LE, TEST, FORLOOP, FORPREP and TFORLOOP.

4.6 Other non-reducible instructions

When encountering instructions like NEWTABLE, these are specific to Lua. Creating a new table is not something that can be reduced with a reasonable benefit to the speed implications to a data-structure in a VM written in Lua. All arithmetic and unary instructions, as well as CONCAT, cannot also be reduced easily, as keeping them in their original form would be much faster

```

local top = 1

local stack = setmetatable({}, {
    __newindex = function(tbl,key,val)
        if key > top then top = key end

        rawset(tbl,key,val)
    end
})

```

Figure 8: The implementation of the stack

than alternatives. Additionally, the equality tests cannot be reduced. Others are simply there to save multiple calls of one instruction (e.g. SETLIST becoming very many SETTABLEs).

After running through all of the different instructions, we have managed to remove usage of several, which should result in a harder time trying to piece together the original program.

5 Creating the Obfuscator

5.1 Creating a bytecode interpreter

Firstly, to form the foundation of the custom VM that will become a part the finished product, a standard bytecode interpreter will be designed. This will just take in standard bytecode as the input, and run it as the output. Parsing the bytecode is simple as there is a defined format for Lua bytecode, so the main thing here to be created is the interpreter itself.

As the obfuscator must have outputs in Lua to work properly, the fact that tables (as lists) have their first value at index 1 means that the bytecode deserialiser must appropriately adapt to this. As a result, after the raw bytecode is converted into a Lua object, another pass-through must be made in order to identify any instructions which need this rectification on their parameters. These are any instructions which deal directly with the constant table (those of format iABx), any instructions which could deal with either a register number or a constant index (RK instructions, where if the value of a parameter B or C has the 8th bit set meaning that it is greater than 256 then the value of the other 7 bits is used as a constant index), any instructions which deal with upvalues, and CLOSURE, which closes a function prototype.

The stack is then implemented. Although it is called a stack, no instructions require popping or pushing a value onto the stack, rather they use indexes to identify values in registers. Only the top of the stack needs to be kept as a variable that needs to be used, and this can be implemented using metatables.

Additionally, as Lua sets all function parameters to the first indexes of the


```

local args = {...}

if #args > 0 then
    for i=0, #args-1 do
        stack[i] = args[i+1]
    end
end
end

```

Figure 9: Setting all values to the stack

stack, all function parameters are obtained using `vararg` and then they are each set as consecutive indices to the stack.

Notice that the stack is indexed from 0 to `#args - 1` here. Although tables in lua are 1-indexed (meaning that if an element is inserted into an empty table using `table.insert`, its index would be 1), setting the 0th item to a table is completely valid as the table data structure encompasses the functionality of lists and dictionaries. As every single instruction that refers to a register number expects the stack to start from index 0 but refers to each register by its explicit index, it is easier to just set the stack like it is a 0 indexed data structure.

As for all of the opcodes, most of them were just stack manipulation and interacting with the upvalue lists, constant lists, the global environment etc. Most of these could be implemented very easily. For calls and returns, they reference the top of the stack, which is where the `top` variable is required.

The most confusing instructions to implement here are those which have a functionality outside of manipulating the stack. This includes the `CLOSE` and `CLOSURE` arguments. Creating closures is relatively easy, as the `compile` function which compiles a function prototype can be recursive. However, setting upvalues needs to also be taken into account. For each upvalue that a closure has, a pseudo-instruction is generated which is either `MOVE` or `GETUPVAL`. These move all of the upvalues into the newly closed closure, and as each compiled function has its own stack, references to the stacks in outer variables must be done through metatable manipulation. As tables are stored as references to an object, passing upvalues with metatable manipulation to index a reference to the outer stack is possible.

5.2 Methods

The aim of the obfuscator is to make it as difficult as possible to take an obfuscated script and turn it back to luac, or even source code. A deobfuscator built to reverse the action of this should have as hard of a time as possible reconstructing. As for the first layer of protection, which is just the fact that it's obfuscated, none of the original source should be visible. This is true as the implementation of a VM taking in bytecode is enough to stump the average developer. To protect against a more dedicated attacker, more must be done.

The chunk must be entirely stripped of any debug information, and anything that could be replaced with something more obscure. All useless debug data can be stripped, like line number information, but in particular, the opcode of each instruction can be removed. A custom VM which generates opcodes only based on what index instruction is currently being processed is a suitable replacement for the removal of opcodes. Therefore, an attacker of this new system would need to additionally parse the VM and match opcode implementations to their actual opcodes. The bytecode must also have a completely different format, so that an attacker can't simply grab the byte string and put that through unluac.

5.3 Serializing a disassembled chunk

As we are designing an obfuscator with protection as the priority, other factors, such as file size, can have less importance. Features like convenience can be completely disregarded as the end user does not require convenience, rather inconvenience to any would-be attacker. In this way we can ignore normal standards when designing the obfuscated VM.

One example of this is the new serialized bytecode. As we do not need to deal with these standards for convenience, usage of IEEE 754 is not necessary. When converting floats we can serialize them as two integers by converting them to a string, splitting it using string methods to remove the decimal point, and then write this as two separate 32 bit integers. This is also useful as to deserialize an IEEE 754 encoded number, it is a more distinct function and therefore easier to spot within obfuscated code. By splitting it into distinctive parts, the implementation of floats and doubles is made easier and hooking a floating point number deserializer in the obfuscated output is made more difficult.

Usage of this however requires specific design implementations. Lua numbers are stored accurately between negative $1e14$ and positive $1e14$, where when they are converted to strings they turn into this exponent form. Additionally, as 64 bit doubles can store up to $DBL_MAX (2^{1024})$, we need to take into consideration that this exponent might also be saved. The number is converted to a string, and then it is parsed to get the exponent, the integral part of the number, and the decimal part. The size of the integers was also increased from 32 to 46 allotted bits (which is 45 useful bits and one sign bit). 46 bits is underneath the $1e14$ boundary, which means that any operations to numbers underneath 2^{46} will behave as expected.

Additionally, with the instructions, as we can deprioritize file size, we can choose to identify each instruction only by its position in the constant list. This means that the final VM will not check the instruction's opcode from 0-37, but it will end up with the opcode implementations for every single instruction. Currently, as the interpreter is comprised of one single interpret function to run the main chunk and any children function prototypes, it is not equipped to handle this. Each function prototype will need to have a certain offset to its first instruction, as if we check by instruction pointer, it will say that the first instruction of the inner prototype is to be run by the opcode of the first

instruction of the outer prototype. Here, a breadth-first search is helpful - we can consider the main chunk a tree graph with children prototypes as nodes. Because these children prototypes can have many more children prototypes and this can recurse down this graph, a breadth-first search allows this offset to be applied to prototypes in an order more relevant than that of a depth-first search. Although this will increase file size, this adds an extra parsing step to any attacker's attack - they cannot just deserialize the bytecode and list off the instructions from there, they need to traverse the main tree of the interpreter loop and then match up each instruction to its opcode.

To create a basic version that works so that progression can be made from there, the upvalue_count will be written, along with the upvalues. This is because if the upvalue count was placed anywhere else, the decoder would assume it is a constant. The upvalues are also strings, so the decoder would assume that they are constants also. Therefore, it is easiest to get these out of the way at the very beginning. Then, the bytecode will be shuffled, but so that when each constant, instruction and function prototype is read, they will be read in order that they were in the original chunk. The decoder in the output will eat a certain amount of bytes to find out what type of object is to be parsed next, and then it will parse it. There will be no set order to what is created, so when the serializer receives the exact same input, it will not necessarily output the same serialized bytecode every time. Each instruction is read as if it has parameters for A, B, C, Bx, and sBx so that no extra information is gained through the parsing of an instruction. The opcode implementation in the compiler will select which parameter to read.

Here is an example of the bytecode for a simple "Hello World" script:

```

101 001 00000000000000000000000000000000 011 00000000000000011100000000
010 001 0000000000000000000000000000000101 01110000011100100110100110111001110100
011 00000001000000000000000000010 010 001 0000000000000000000000000001100
0100100001100101011011000110110001101111001000000101011101101110111001001101100011001000010000
011 0000000000000000010000000001 011 000000000000000001000000000

```

Figure 10: A colorcoded segment of the bytecode of the Hello World script.

5.4 Breaking down the instructions

To break down the instructions as specified earlier, a pass through is made to insert a new set of instructions into a different table. However, any jumps in the instruction list need to take into account the fact that a single instruction could have become many different instructions, resulting in these jumps not being correct. Therefore, a preliminary pass-through must be made before the actual breaking down of instructions, checking the referenced instruction that the JMP would target. Then, during the actual pass-through, the instruction index that the new instructions point to would be specified for each instruction, so that a final pass-through can be made to correct the actual sBx parameter. After these

corrections have been made, we are left with a chunk that does exactly what the original chunk does, but is comprised of different instructions so that any attacker will have to match these new instructions to their old functionalities, all without any visible difference to the end user on the outside.

5.5 Generating the VM

Due to already having created an interpreter earlier, the opcode implementations can be recycled for this part. Additionally, the main body of the compile function, with the stack definitions and all the parameters set, can be recycled as well. When implementing protos, the instruction offset of each prototype must be encoded in the bytecode. This will be an unsigned integer, so it is easiest to just encode it next to the `upvalue_count`. This allows the compiler to get the instruction offset for each prototype and adjust the instruction pointer accordingly. The order of each chunk's probable execution has already been obtained through the breadth-first search conducted earlier to offset each chunk by a certain amount, so we can iterate through this order and construct an if-else tree for each instruction inside. The if statements will compare the instruction pointer as stated earlier.

Additionally, anti-tamper functions are added to prevent the inexperienced attacker from beautifying the script and being able to still run it. These use `debug.lineinfo` and `pcall` to get the running line number, and check to see if the script is still on the same line. Although error messages from `pcall` do not follow the same format, the error message format of Lua and LuaJIT are similar enough that one single `string.find` operation is able to accurately access the line number, which is provided in the error message. If the line numbers do not match, the script will be crashed with a `while true do end` loop.

All constants and necessary functions for the execution of the VM are then appended to the beginning, as well as the actual bytecode. Then, this output is put through a minifier, specifically `luasrcdiet` [2].

Finally, obfuscation has been achieved. Figure 11 shows the output for `print("Hello from my new VM!")`:

5.6 Structure of the output

All required environment functions for the running of the obfuscator are given new names as locals, which is done in the first part of the script (from the local `o="ldexp"` to `t="seed"`). This allows the minifier to call those functions without referencing their defined names, so that when inspecting the VM it is hard to see what environment functions are called. Next, functions required for the use of the VM which are not in the environment are defined. This includes a function to find a value in a table, a function that will error (for the use of the line check functions), 2 line check functions using different methods, and utility functions for the bytecode deserializer. Then the bytecode is defined (the local `A`). The deserializer is then defined, which will break the bytecode down into something that the VM can read. This is from the local `V`, which stores the bytecode flags,

Figure 11: The exact output of the obfuscator. A glot.io snippet is available at <https://glot.io/snippets/gp4eabrh15>

13

in the parameter `d` of `r`. The stack is initialised (`stack = o`), and then after the `while true do` loop, the main instruction tree follows. Each instruction has an enum `l`, which is the instruction pointer, and this matches each instruction with the actual block which executes the instruction as its allocated opcode. Finally, the last statement, `return r(y(A))()`, runs the VM, translating to `return compile(decode(bytecode))()`.

6 Conclusion

Overall, the aims of this project were fully met. I was able to dive deeper into the internals of Lua, and use that knowledge to create a functioning obfuscator. In the future, I hope to improve the security of the obfuscator, as all of the constants are very exposed and every single part of the output is useful to an attacker - there is nothing to throw them off the scent.

7 Extras

```
local function toBenchmark(POT)
    local total = 0
    for i=1, 10 ^ POT do
        total = total + i^2
    end

    return total
end

print("POT\t|time taken")

for i=6, 10 do
    local start = os.clock()

    toBenchmark(i)

    local taken = os.clock() - start
    print(string.format("%s\t| %s", i, taken))
end
```

Figure 12: Lua code used for benchmarking, section 1

```

import time

def toBenchmark(POT):
    total = 0
    for i in range(10 ** POT):
        total += i ** 2

    return total

print("POT\t|time taken")

for i in range(6,11):
    start = time.perf_counter()

    toBenchmark(i)

    taken = time.perf_counter() - start
    print(f"{i}\t|{taken}")

```

Figure 13: Python code used for benchmarking, section 1

References

- [1] The Lua Authors. Lua. <https://www.lua.org/source/5.1/>. Accessed: 16-05-2023.
- [2] Jakub Jirutka Watson Song Peter Melnichenko Kein-Hong Man. luasrddiet. <https://github.com/jirutka/luasrddiet>. Accessed: 27-07-2023.
- [3] Roblox. Luau. <https://github.com/roblox/luau>. Accessed: 16-05-2023.
- [4] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society.*, 43, 1938.