Lua bytecode obfuscation

The idea behind my personal Obfuscator,alpine

By Joseph/The Wagoner

Credits

The Wagoner/Me - Coming up with this idea in general, and being the one who made this, My v3rmillion: Here

khman- Shout out khman for Chunkspy and a no frills guide to lua 5.1 vm instructions, which is basically what helped me make this project

Louka - Made the original documents that inspired this one, Sadly louka passed away after being thrown out of a helicopter on 2/31/21

My buddies over at lynx- For being cool

Note: I may have not been the first person to have this idea, honestly i would be surprised if im the first, but I came up with it independently of any others so SUCK IT losers

Project premise

My project, Alpine, was born off the backs of school boredom, Started in school while bored as hell. It has been worked on for ~2 months now, as I this project is what made me understand Lua instructions *(aswell as machine/bytecode instructions)* as a whole.

The General premise of this project was to take lua bytecode, and produce the same program but with scrambled control flow, replaced instructions (*C. Polymorphic code*), et cetera.

This entire goal was accomplished by forming an engine that abstracts Lua bytecode in a very particular way that makes modification easier.

I may say something stupid in this document and if I do, excuse me, as this entire project is a learning experience for me.

Specifications of my engine

The basic premise of the engine I've developed is to take lua bytecode and through ~2 short steps, abstract it into something more

Parsing
Index modification

Parsing

- Forms the bytecode Into an object, easy for modification, *(EXAMPLE)*

Index modification

- Takes formerly number based indexes, like constants, protos, etc and transforms them into uuid based indexes,
- Also gives every instruction a uuid, and replaces jmp instructions with the id of the instruction they jmp to

Assembly

Following the Parsing stage, You may ask, how do we re assemble this, well basically just reverse the process,

Look Through all instructions, Re-Replace the uuids with indexes,
Reconstruct the given original object back into bytecode

These two steps are fairly simple, as it is basically the same as just replacing the uuids with indexes, and then using a lookup table for what index to replace with, and reconstruction just reverses the original parsing process by taking the given table and constructing it back into instructions.

How does this help with obfuscation?

You may be scratching your head as to how this can help with obfuscation, well it can actually be incredibly useful in plethora of ways

Let me give you a hypothetical situation, Say we have an array of cells, and cell 3 contains information for a user, lets call him bob

```
        1    2    3    4    5
Cells: [39] [13] [10] [69] [42]
Bob_Info_reference: 3
```

So we know that bobs information is held a cell 3, with a value of 10, so what if we need to add a new cell at position 2?

```
        1    2    3    4    5    6
Cells: [39] [40] [13] [10] [69] [42]
Bob_Info_reference:3
```

The number that points to his info is still 3, but that is wrong, as his info has shifted up 1, and is now contained in the fourth cell
Well this is the simple idea and reason that we need to use uuids instead of number indexes (This is pretty simple stuff, just felt like explaining)

A more related example

Since the last example was easy to understand for new people, Heres a version that might be more comprehensible for experienced people.
Constants = {1,”hello”}
Instructions = {
	Push r0,k1 -- push “hello” into register 0
	Call print, r0 -- call print using argument in register 0
}

So what happens if we add a new constant, for example:3, while not modifying our instructions
Constants = {3,1,”hello”}
Instructions = {
	Push r0,k1 -- push 1 into register 0
	Call print, r0 -- call print using argument in register 0
}
Since we have modified the constant list, while not modifying our instructions, the behavior of the program has changed, in a undesirable way

Types of viable modification

So this entire engine I've developed is pretty sick, and allows for all the following types of obfuscation

- Control flow scrambling (random jmps littered throughout the program
- Anti-Decompilation code to be inserted
- Easy instruction modification (Replacing loadk and such with more complexity)
- Replacing globals as a whole with new names, and or a whole new medium for them to be loaded/stored
- Removing the use of rk() as a whole as to allow for better obfuscation
- So so much more

Some of these modifications are short, and arent that special (For ex anti decompilation code, which in my current program is literally entering a string with no null terminator '\0' after it to the constant list, which breaks unluac)

Control flow

The specifics of the control flow “obfuscation” in my program is not complicated, it doesnt use any kind of analysis to detect ifs, whiles, etc etc as it only modifies the natural flow of execution, a quick example of this program would be something like

Normal program for ‘for i=1,100 do print(i) end’

```
[1] loadk     0   2       ; 1
[2] loadk     1   0       ; 100
[3] loadk     2   2       ; 1
[4] forprep   0   3       ; to [8]
[5] getglobal 4   1       ; print
[6] move      5   3
[7] call      4   2   1
[8] forloop   0   -4      ; to [5] if loop
[9] return    0   1
```

Control flow Modified version

```
[01] loadk     0   1      ; 1
[02] jmp       2          ; to [5]
[03] getglobal 4   2      ; print
[04] jmp       7          ; to [12]
[05] jmp       2          ; to [8]
[06] loadk     2   1      ; 1
[07] jmp       2          ; to [10]
[08] loadk     1   3      ; 100
[09] jmp       -4         ; to [6]
[10] forprep   0   3      ; to [14]
[11] jmp       -9         ; to [3]
[12] move      5   3
[13] call      4   2   1
[14] forloop   0   -4     ; to [11] if loop
[15] return    0   1
```

Tiny note: This modifies decompiled output heavily

Constant loading

I will try to keep this slide short as its not very complex

So I had noticed something pretty simple about arithmetic instructions in lua, They can be used in the same way that loadk can be used, they can load constant values into registers just like loadk is used for numbers,

So I decided to use this simple idea for something in my obfuscator, And the result is pretty good and can be used very effectively with some other modifications (Removal of rk in favor of extra instructions)

For ex, Instructions like:
loadk 0,k(3); k signifies a constant,
Can be replaced with:
mul 0,k(0.75),k(4)
And be exactly the same type of code

Pros/Cons of this type of obfuscation

There are many bad things that come with this obfuscation

Cons:

- Compatibility issues with environments that dont have bytecode loading support
- No vm used in this obfuscation, so a deobfuscator could be made ~fairly easily (Vm support maybe soon)

Pros:

- Allows for an easy custom compiler to be made
- Loads faster due to the compiler not being needed
- Can be used with much effect in a vm if implemented properly
- Hard to reverse for newbs

Closing

This project may come to market eventually but I am not sure, if you are interested in this project you can add me at: Wag#0001

IF YOU HAVE ANY EXPERIENCE IN VM GENERATION AND MAY WANT TO HELP ME, YOU CAN ADD ME AT Wag#0001, YOUR HELP MAY BE APPRECIATED

Thank you and have a great day
-Wag