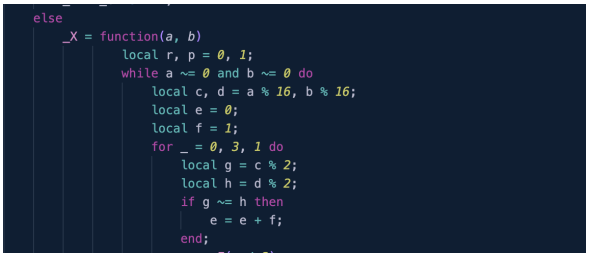
I get this question a lot. Someone DMs me saying they ran our protected script through some deobfuscator and it spat out garbage.

So let me actually explain it.
Most static deobfuscators work by recognizing patterns in standard Lua 5.1 bytecode. They know what `OP_ADD` looks like. LuaVirtualBox doesn't use standard Lua bytecode.
When your script gets compiled, it gets translated into a completely custom instruction set with a randomized opcode mapping. Opcode `0x05` might be ADD in one file and MOVE in the next.



```
else
    _X = function(a, b)
            local r, p = 0, 1;
            while a ~= 0 and b ~= 0 do
                local c, d = a % 16, b % 16;
                local e = 0;
                local f = 1;
                for _ = 0, 3, 1 do
                    local g = c % 2;
                    local h = d % 2;
                    if g ~= h then
                        e = e + f;
                    end;
```

```
-- Simple example script
local function add(a, b)
    return a + b
end

local function greet(name)
    local message = "Hello, " .. name .. "!"
    print(message)
end

local sum = add(10, 20)
print("The sum is: " .. sum)
greet("World")
```

A static tool opens the output and stares at a binary blob with no frame of reference. The only way to understand it is to reverse-engineer the interpreter embedded *specifically for that file* which is a completely different problem.

2. The Control Flow Is Encrypted
A lot of VM-based obfuscators use a big dispatcher loop a `while` with a `switch` inside. Static analyzers have gotten good at following these.

We don't use a switch statement.
Instead, the VM uses Blob Dispatching. The information about which block executes next is stored in an encrypted binary blob. At runtime, the VM decrypts the next instruction's location using a rolling key derived from the current execution state.

```
local ix = (function()
    local _s = 690081577;  -- Blob seed
    ...
    return function(_B, _id)  -- Encrypted block lookup
```

From a static analyzer's perspective, it sees a `while` loop calling a function pointer. That's it. The path through the program is invisible until the code is actually running.

3. Constants Are State-Dependent
In most obfuscated Lua, there's still a constant table somewhere. strings, numbers, maybe XOR'd with a static key. A tool can just extract them.
Not here. Every constant is retrieved through a decoder that requires a key and seed that depend on the VM's internal state at that exact moment. The key used to decode a string at instruction 50 might be derived from what happened at instruction 10.

A static tool isn't running the code. It has no idea what the VM state is. You'd need a perfect, cycle-accurate simulation of the entire execution at which point you've basically written a full emulator.

4. Even Lifted Code Is Unreadable
Let's say someone is very patient and partially lifts the bytecode back into something resembling logic. They're still not done.

Within each basic block, instructions that don't have data dependencies on each other are shuffled into a random order. The semantics are preserved. the code still does the same thing, but the *structure* is destroyed.

```
-- Block 0xA3F1 (dispatched randomly via encrypted blob)
function()
    k = E[14](t)          -- unknown op: what is op 14? what is t?
    M = E[12](<k, U>)       -- k == U, but U is not yet visible here
    M = 6230468           -- next block ID (hidden in encrypted blob)
end,

-- Block 0x5B2C (executes BEFORE 0xA3F1 per blob, but listed AFTER in table)
function()
    U = A("9/", 2, 1328085341, 4196150, ...)  -- "10" encrypted
    S = A("_a", 2, 1613549023, 4196150, ...)  -- "0" encrypted
    k = E[9](t)           -- unknown op applied to unknown register
    M = E[12](<k, U>)       -- comparison, but operands are opaque
    M = 6230468           -- next block ID
end,

-- Block 0xD901 (contains the actual loop body)
function()
    q = q + k             -- count = count + 1
    Z = not j             -- opaque predicate noise
    p = q <= m            -- count <= max?
    p = Z and p
    Z = q >= m            -- count >= max?
    Z = j and Z
    p = Z or p
    Z = A("_i$L...", 2, 1645199001, 5521796, ...)  -- next block if true
    M = p and Z
    p = A("_2_z...", 2, 1916805715, 5521796, ...)  -- next block if false
    M = M or p            -- M = next block ID (conditional dispatch)
end,

-- Block 0x1177 (poison/fake block — never actually reached)
function()
    local R = 51
    M = "game.Players.LocalPlayer.Kick"   -- honeypot trap
    v = "Prometheus.Trace.Noise"          -- honeypot trap
```

end,