Mitigating the Danger of Malicious Bytecode

Peter Cawley <lua@corsix.org>

Lua Workshop 2011

A Common Pattern

1. Create sandbox
2. Load user-supplied Lua (byte|source) code
3. Run code in sandbox

A Common Pattern

1. Create sandbox
2. Load user-supplied Lua (byte|source) code
3. Run code in sandbox

Sandbox Blacklist

```
     os.*
     io.*
  debug.*
package.loadlib
package.loaders[3]
package.loaders[4]
```

A Common Pattern

1. Create sandbox
2. Load user-supplied Lua (byte|source) code
3. Run code in sandbox

Sandbox Whitelist

```
string.gsub
 table.sort
```

A Common Pattern

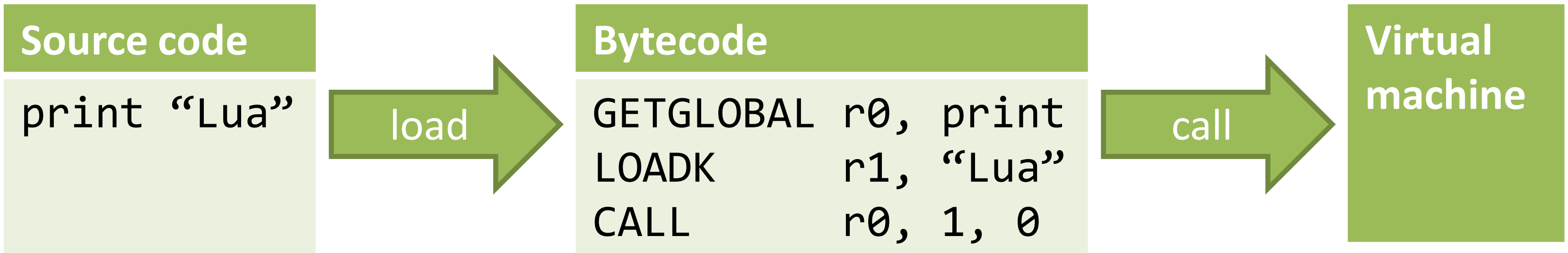
1. Create sandbox
2. Load user-supplied Lua (byte|source) code
3. Run code in sandbox
   - Arbitrary native code execution*

Sandbox Whitelist

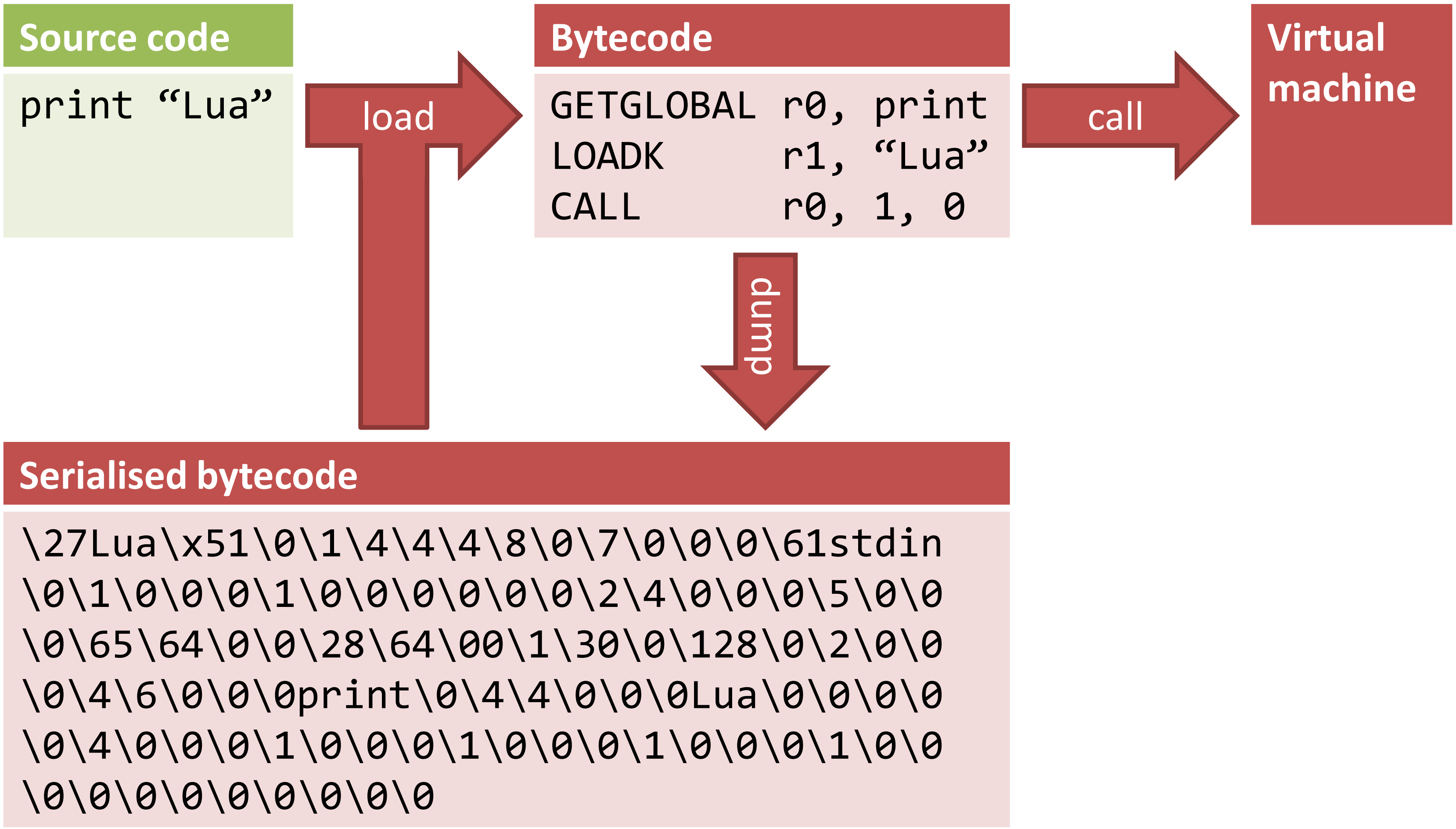
```
string.gsub
 table.sort
```

* At least for Lua 5.1.4 on x86 Windows (even with DEP and ASLR)

Bytecode

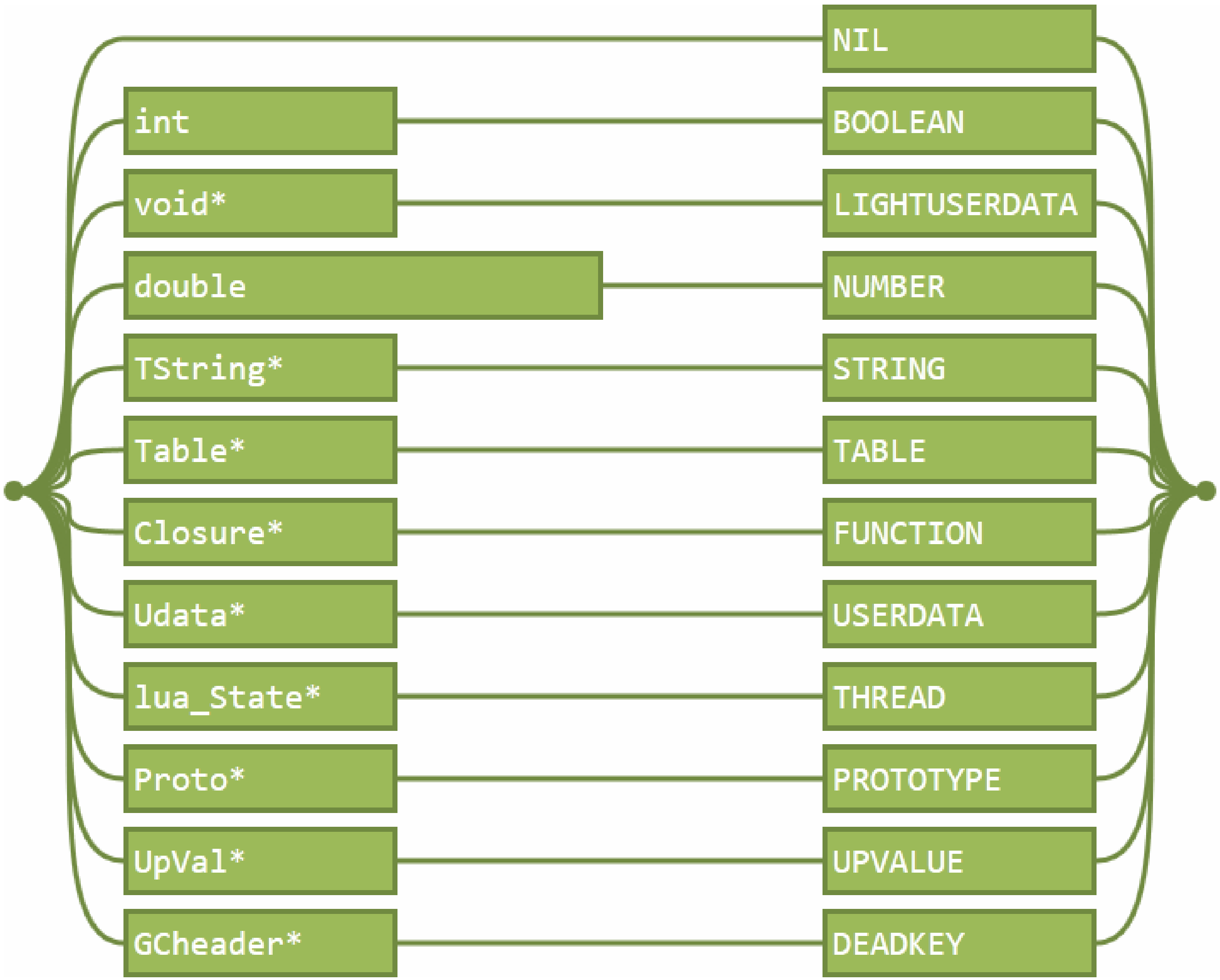

Source code
print “Lua”
load
Bytecode
GETGLOBAL r0, print
LOADK r1, “Lua”
CALL r0, 1, 0
call
Virtual machine

Bytecode

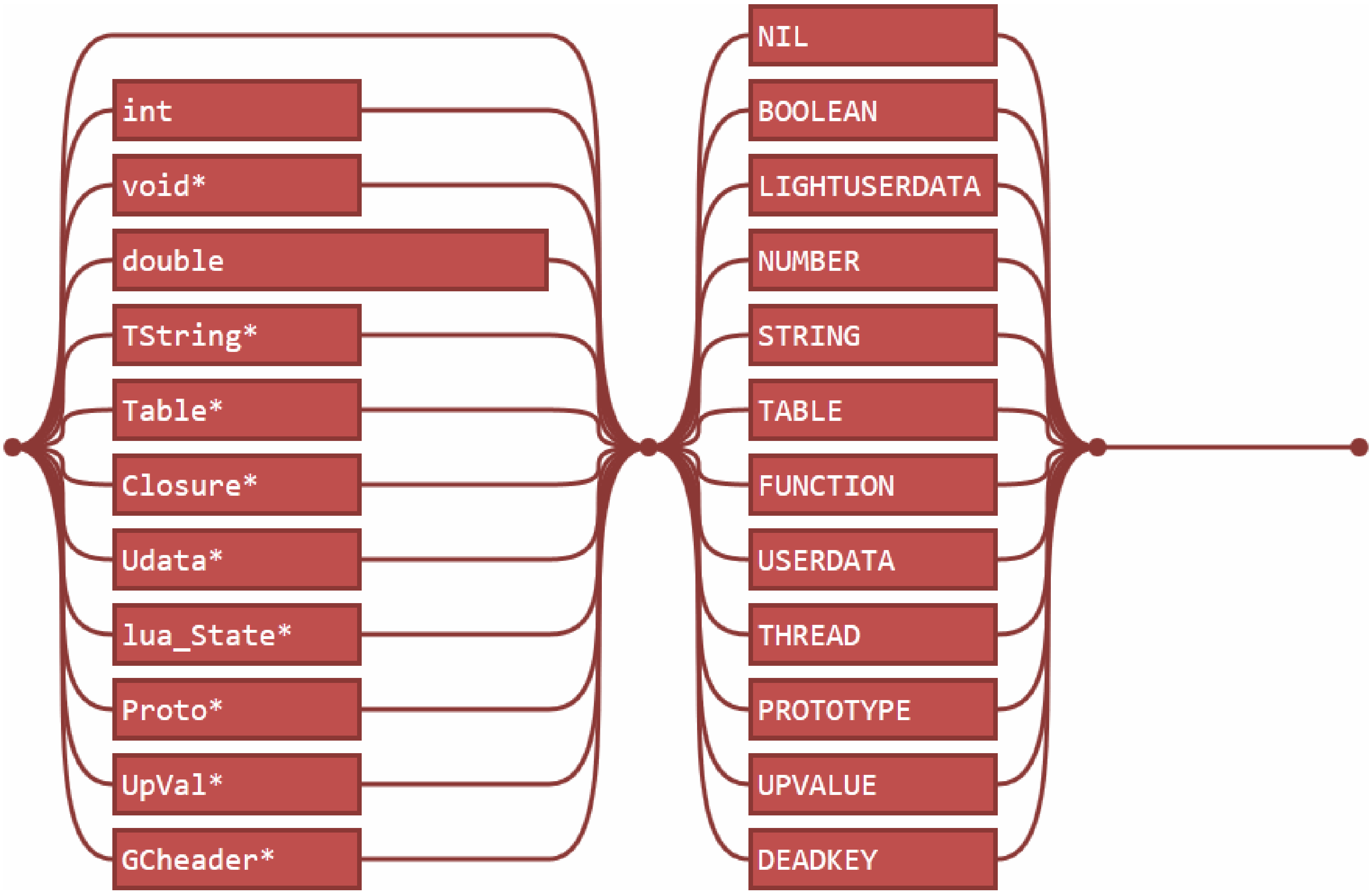


Source code
print “Lua”
load
Bytecode
GETGLOBAL r0, print
LOADK r1, “Lua”
CALL r0, 1, 0
call
Virtual machine
dump
Serialised bytecode
\27Lua\x51\0\1\4\4\4\8\0\7\0\0\0\61stdin
\0\1\0\0\0\1\0\0\0\0\0\0\2\4\0\0\0\5\0\0
\0\65\64\0\0\28\64\00\1\30\0\128\0\2\0\0
\0\4\6\0\0\0print\0\4\4\0\0\0Lua\0\0\0\0
\0\4\0\0\0\1\0\0\0\1\0\0\0\1\0\0\0\1\0\0
\0\0\0\0\0\0\0\0\0

Logical TValue

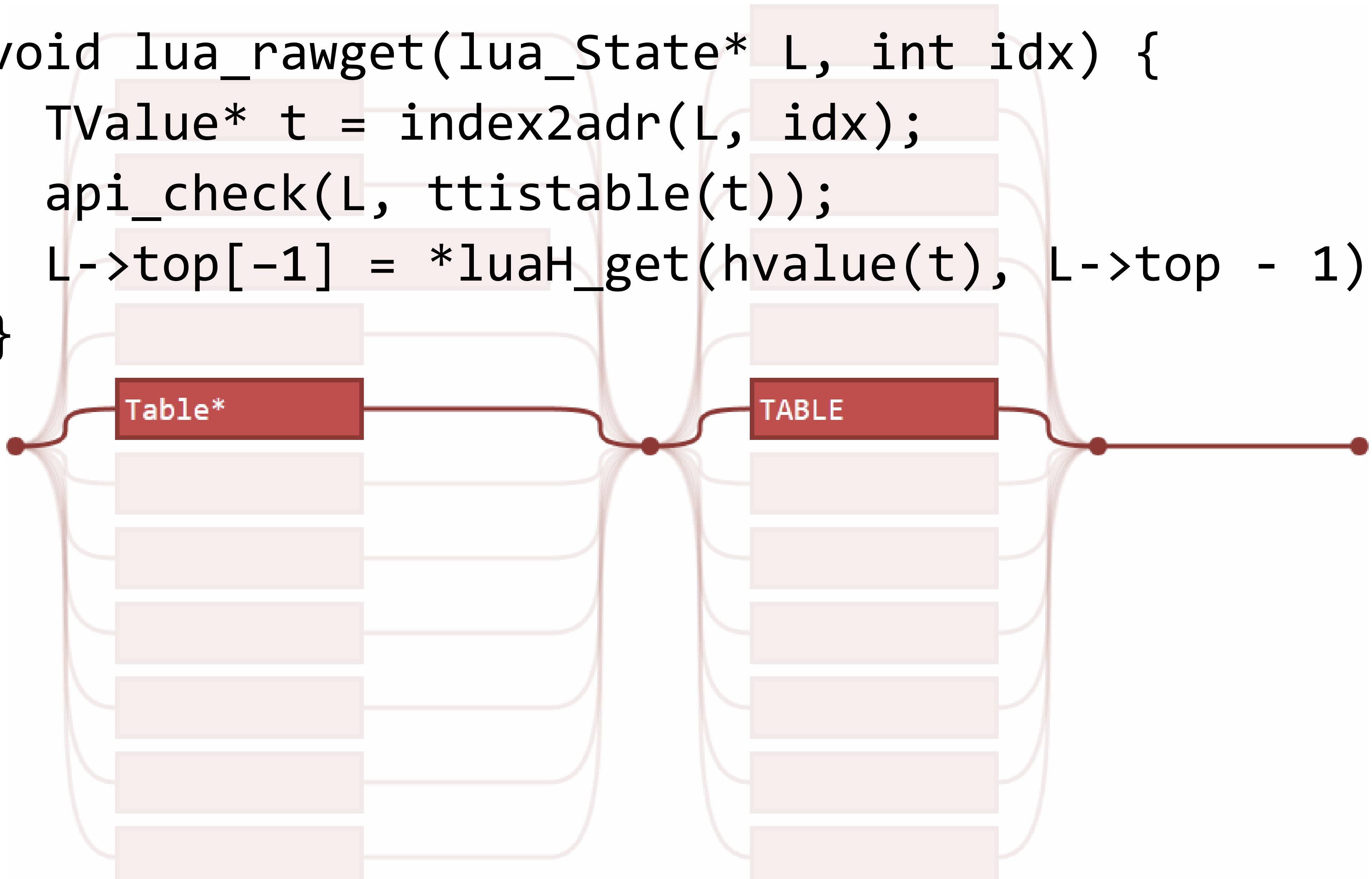


Physical TValue



C API abusing a TValue

```
void lua_rawget(lua_State* L, int idx) {
  TValue* t = index2adr(L, idx);
  api_check(L, ttistable(t));
  L->top[-1] = *luaH_get(hvalue(t), L->top - 1);
}
```



C API abusing a TValue

```
void lua_rawget(lua_State* L, int idx) {
  TValue* t = index2adr(L, idx);
  api_check(L, ttistable(t));
  L->top[-1] = *luaH_get(hvalue(t), L->top - 1);
}
```


```
int table.sort(lua_State* L) {
  luaL_checktype(L, 1, LUA_TTABLE);
  /* ... */
  lua_rawget(L, 1);
  /* ... */
}
```

C API abusing a TValue

```
void lua_rawget(lua_State* L, int idx) {
  TValue* t = index2adr(L, idx);
  api_check(L, ttistable(t));
  L->top[-1] = *luaH_get(hvalue(t), L->top - 1);
}
```


```
int table.sort(lua_State* L) {
  luaL_checktype(L, 1, LUA_TTABLE);
  /* ... call comparison function ... */
  lua_rawget(L, 1);
  /* ... call comparison function ... */
}
```

Virtual Machine abusing a `TValue`

```
for x = init,
        limit,
        step
do
  print(x)


end
```

```
GETGLOBAL init
GETGLOBAL limit
GETGLOBAL step
FORPREP
GETGLOBAL print
MOVE x
CALL
FORLOOP
```

Function Calls

```
local t = {“go”, “a”}
table.sort(t,
  function(lhs, rhs)
    return #lhs < #rhs
  end)
```

Function Calls

```
local t = {"go", "a"}
table.sort(t,
  function(lhs, rhs)
    return #lhs < #rhs
  end)
```

{"go", "a"}	r0
table.sort	r1
{"go", "a"}	r2
function	r3
	r4
	r5
	r6
	r7
	r8
	r9

Function Calls

```
local t = {“go”, “a”}
table.sort(t,
  function(lhs, rhs)
    return #lhs < #rhs
  end)
```

{“go”, “a”}	
table.sort	
{“go”, “a”}	1
function	2

Function Calls

```
local t = {“go”, “a”}
table.sort(t,
  function(lhs, rhs)
    return #lhs < #rhs
  end)
```

{“go”, “a”}	
table.sort	
{“go”, “a”}	1
function	2
“go”	-5
“a”	-4
function	-3
“a”	-2
“go”	-1

Function Calls

```
local t = {“go”, “a”}
table.sort(t,
  function(lhs, rhs)
    return #lhs < #rhs
  end)
```

{“go”, “a”}	
table.sort	
{“go”, “a”}	
function	
“go”	
“a”	
function	
“a”	r0
“go”	r1
	r2

Function Calls

```
local t = {“go”, “a”}
table.sort(t,
  function(lhs, rhs)
    return #lhs < #rhs
  end)
```

{“go”, “a”}	
table.sort	
{“go”, “a”}	
function	
“go”	
“a”	
function	
“a”	r0
“go”	r1
false	r2

Function Calls

```
local t = {“go”, “a”}
table.sort(t,
  function(lhs, rhs)
    return #lhs < #rhs
  end)
```

{“go”, “a”}	
table.sort	
{“go”, “a”}	1
function	2
“go”	-3
“a”	-2
false	-1
“a”	
“go”	
false	

Function Calls

```
local t = {“go”, “a”}
table.sort(t,
  function(lhs, rhs)
    return #lhs < #rhs
  end)
```

{“a”, “go”}	
table.sort	
{“a”, “go”}	1
function	2
“go”	
“a”	
false	
“a”	
“go”	
false	

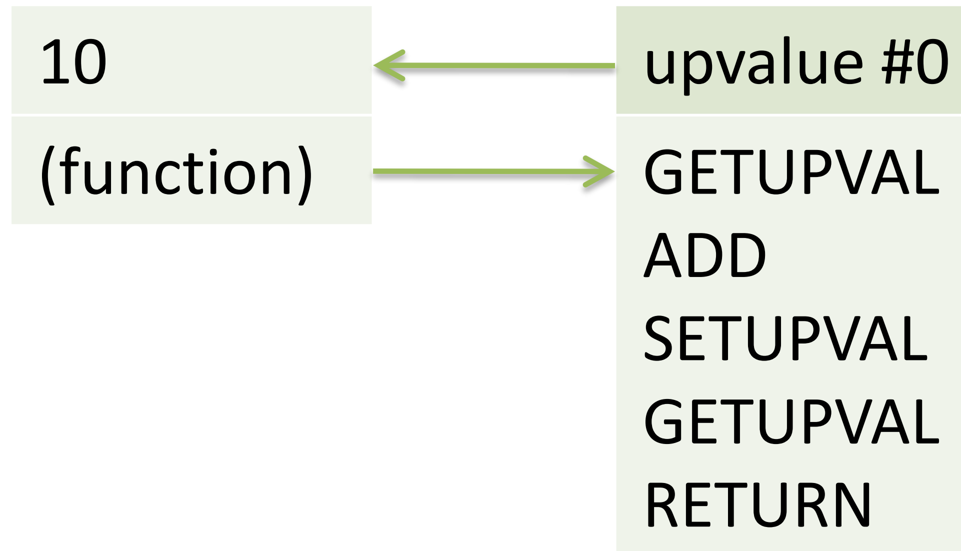
Function Calls

```
local t = {“go”, “a”}
table.sort(t,
  function(lhs, rhs)
    return #lhs < #rhs
  end)
```

{“a”, “go”}	r0
table.sort	r1
{“a”, “go”}	r2
function	r3
“go”	r4
“a”	r5
false	r6
“a”	r7
“go”	r8
false	r9

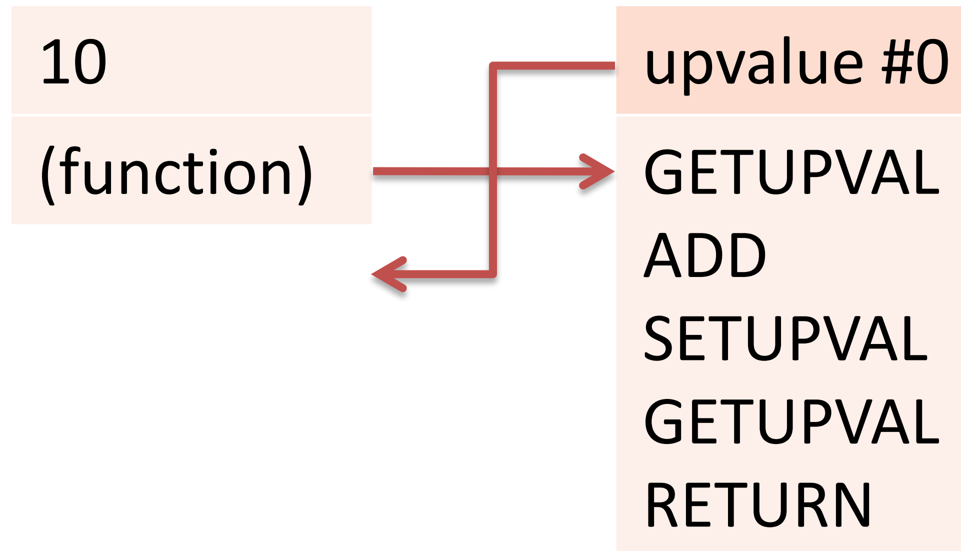
Upvalues

```
local x = 10
local count =
  function()
    x = x + 1
    return x
  end
```



Upvalues

```
\27Lua\x51
...
(malicious
 bytecode
 here)
...
```



Malicious Bytecode Catalogue

- Violating type assumptions in the VM
  - `FORLOOP`
  - `SETLIST` in 5.2
- Emulating `debug.[gs]etlocal`
  - Reading leftover locals
  - Promiscuous upvalues
- Violating type assumptions in the C API
  - `lua_(next|raw[gs]eti?)`
  - `lua_[gs]etuservalue` in 5.2

Mitigation Catalogue

- Don’t load bytecode
  - First byte decimal 27
  - `load(ld, source, "t" [, env])` in 5.2
- Compile with `LUA_USE_APICHECK` (*)
- Static analysis and verification of bytecode

(*) Makes exploitation harder, doesn’t prevent information leakage attacks, may not save you.

Static Analysis, Blunt Approach

- Violating type assumptions in the VM
  - For each stack slot, at each VM instruction, determine a set of possible types
- Emulating `debug.[gs]etlocal`
  - Ensure stack slots are safely readable
  - For each stack slot, at each VM instruction, determine if it could be an upvalue
  - Segregating calls from upvalues

Static Type Analysis

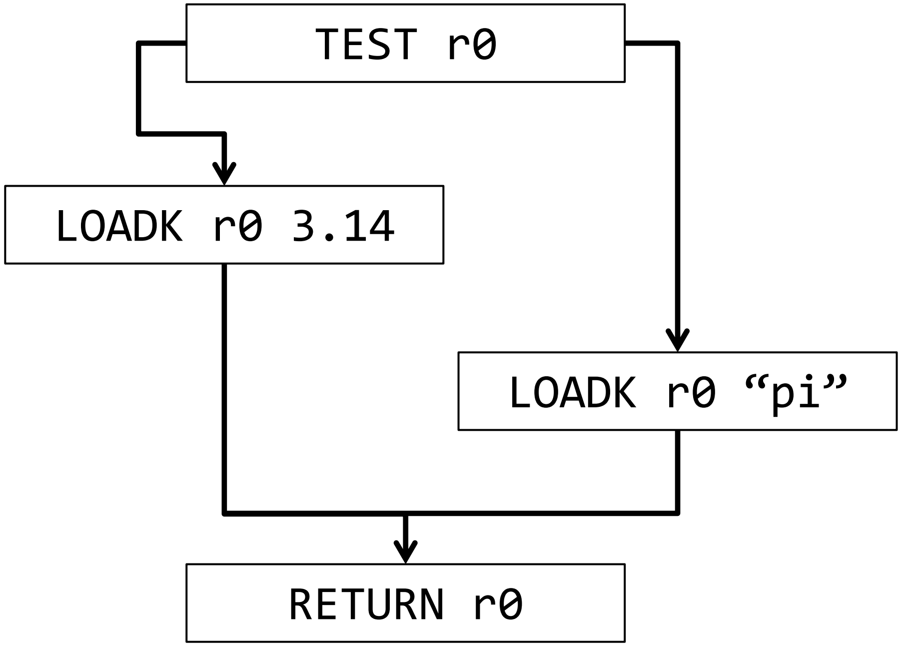
```
function example(x)
  if x then
    x = 3.14
  else
    x = “pi”
  end
  return x
end
```

```
.parameter r0
TEST r0; JMP $+2
LOADK r0, k0
JMP $+1
LOADK r0, k1

RETURN r0
```

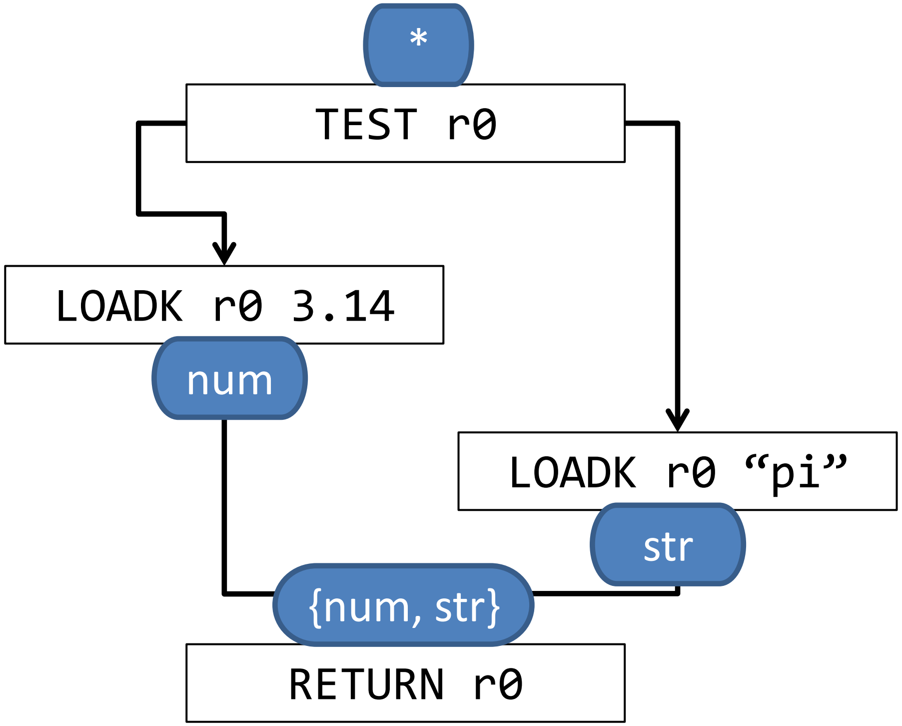
Static Type Analysis

```
function example(x)
  if x then
    x = 3.14
  else
    x = “pi”
  end
  return x
end
```



Static Type Analysis

```
function example(x)
  if x then
    x = 3.14
  else
    x = “pi”
  end
  return x
end
```



Static Analysis Prerequisites

- Decode and understand each instruction
- Ensure control flow doesn't leave
- Valid (register|constant|…) indices
- Verify some VM assumptions, like:
  - TEST instructions are followed by a `JMP`
  - Boolean constants are either 0 or 1
- Instructions which produce or consume a variable number of values must come in pairs

Static Analysis, Subtle Approach

- Violating type assumptions in the VM
  - Protect loop control variables
  - Perform runtime table type checks
- Emulating `debug.[gs]etlocal`
  - At each VM instruction, split the stack into locals / temporary / unused

Upvalues Calls Unreadable

Static Analysis, Subtle Approach

- Debug information embedded within bytecode
  - Gives size of the local region at each instruction
  - Specifies which locals are loop control variables
- The temporary region always grows into the next available unused stack slot
- The local region always grows to absorb a temporary
- Backward jumps are to locations with no temporaries
- Forward jumps merge to the smallest of the temporary ranges

“Practical” Static Analysis

```
require "lbcv"

lbcv.verify(ld)

lbcv.load(ld [, source [, mode]])
```

Questions?

Peter Cawley <lua@corsix.org>

Lua Workshop 2011