Reverse Engineering of Obfuscated Lua Bytecode via Interpreter Semantics Testing

Chenke Luo, Jiang Ming, *Member, IEEE,* Jianming Fu, *Member, IEEE,* Guojun Peng, *Member, IEEE,* Zhetao Li, *Member, IEEE*

***Abstract*—As an efficient and multi-platform scripting language, Lua is gaining increasing popularity in the industry. Unfortunately, Lua's unique advantages also catch cybercriminals' attention. A growing number of IoT malware authors switch to Lua for malicious payload development and then distribute malware in bytecode form. To impede malware code analysis, malware authors obfuscate standard Lua bytecode into a customized bytecode specification. Only the attached interpreter can execute that particular bytecode file. Rapid recovery of Lua obfuscated bytecode is essential for a swift response to new malware threats. However, existing generic code deobfuscation approaches cannot keep up with the pace of emerging threats. In this paper, we present a novel reverse engineering technique, called *interpreter semantics testing*. Given a customized interpreter used to execute obfuscated Lua bytecode, we construct a set of *LuaGadgets* that can adapt to the customized interpreter. Each LuaGadget contains a carefully chosen opcode sequence to fulfill an observable calculation—it is designed to test one or two particular opcodes at a time. Next, we mutate unknown opcode values to generate a bunch of test cases and run them using the customized interpreter; we can observe the expected result only when the mutation hits the opcode's right value. We perform test case prioritization to cost-effectively recover the semantics of all obfuscated opcodes. Our approach makes no assumptions about the interpreter's structure and is free from analyzing the numerous execution traces of opcode handlers. We have evaluated our tool, *LuaHunt*, with Lua malware variants and real-world applications. LuaHunt is able to recover the obfuscated bytecode's semantics within 90 seconds for each test case, and all of our deobfuscation results can pass the correctness testing. The encouraging results demonstrate that LuaHunt is a promising tool to lighten the burden of security analysts.**

***Index Terms*—Malware Analysis, Bytecode Obfsucation, Deobfuscation, Lua, Interpreter Semantics Testing.**

I. Introduction

ACCORDING to the PYPL (PopularitY of Programming Language) Index [1], Lua's popularity is on the rise in recent years. Lua has risen as one of the best programming languages for modern game engines, such as Angry Birds and World of Warcraft [2]. Many applications utilize Lua as their primary embedded script language to support dynamic script extensions, which include but are not limited to Photoshop, Apache, NetBSD, and Wireshark [3]. As shown in Figure 1, Lua interpreter can load and execute Lua programs either in textual source form (Figure 1(a)) or in precompiled bytecode form (Figure 1(b)). The advantages of running precompiled Lua bytecode are faster loading and protecting source code from intentional or accidental changes.

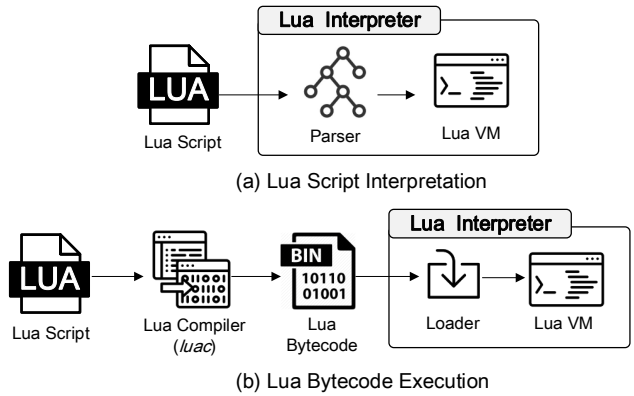

Figure 1. Running a Lua program from scripts or bytecode. By contrast, the interpreter is written in C and distributed in machine code form.

However, it never takes malware authors long to catch up with the latest technique trend. Since 2016, we have observed that the number of IoT malware developed using Lua was increasing [4]–[16], which makes malware development much easier and malware distribution more flexible in miscellaneous embedded devices. For example, LuaBot is the first IoT Botnet written in Lua language and had a zero-detection rate when it was found [4]. LuaBot's functional modules are programmed as multiple ".lua" source files, along with the Lua runtime libraries. LuaBot author acknowledged that the lightweight, cross-platform features and the easiness to integrate with C code were the primary reason to choose Lua [5].

Another representative example is GodLua, which is the first-ever malware spotted abusing the new DoH (DNS over HTTPS) protocol to hide its DNS traffic [7]. GodLua downloads and runs Lua bytecode files to launch HTTP flood attacks targeting some websites. GodLua evolves from its predecessor, which was written in C and can only run on x86/x64 platforms. In contrast, GodLua now supports multiple platforms (x86/x64, ARM, and MIPS) [17]. Furthermore, to frustrate reverse engineers and malware researchers, GodLua obfuscates the bytecode file by randomizing the bytecode file format and bytecode instruction set (i.e., opcode); it attaches a modified interpreter to execute the new bytecode file. The bytecode

This work was supported in part by the National Key R&D Program of China (2021YFB3101201), the National Natural Science Foundation of China (61972297, 62172308, 62172144).

Chenke Luo, Jianming Fu, and Guojun Peng are with the School of Cyber Science and Engineering, Wuhan University, Wuhan, Hubei, 430072 China (e-mail: kernelthread@whu.edu.cn, jmfu@whu.edu.cn, guojpeng@whu.edu.cn).

Jiang Ming is with the Department of Computer Science, Tulane University, New Orleans, LA 70118 USA (e-mail: jming@tulane.edu).

Zhetao Li is with the National & Local Joint Engineering Research Center of Network Security Detection and Protection Technology, Guangdong Provincial Key Laboratory of Data Security and Privacy Protection, College of Information Science and Technology, Jinan University, Guangzhou 510632, China (e-mail: liztchina@hotmail.com).

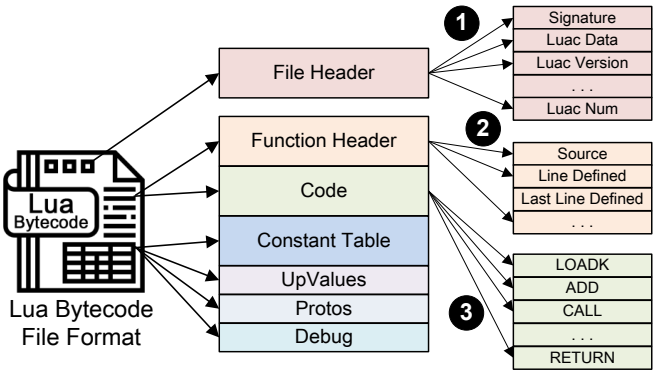

Figure 2. The data structure of a Lua bytecode file.

and interpreter can vary significantly from one obfuscated instance to the other, preventing the generation of reliable malware signatures. Lua decompiler (e.g., LuaDec [18]) and Lua code static analyses [19], [20] are rendered useless on such programs, and only the interpreter's binary code is directly readable.

This paper takes the first step towards the rapid recovery of Lua obfuscated bytecode, which is essential for defeating the emerging Lua malware. Lua bytecode obfuscation bears a striking resemblance to code virtualization obfuscation [21]–[24], which transforms the original instruction set architecture (ISA) into bytecode in a new ISA format. At runtime, the bytecode is emulated by an embedded interpreter on the real machine. Although many techniques have been proposed to deobfuscate the virtualized code, they are unsuitable for handling Lua obfuscation. As the classic bytecode interpreter structure is the so-called "decode-dispatch loop" [25], one category of previous works detects such decode-dispatch loops and finds the mappings between opcodes and related handler functions [26]–[29]. However, the latest Lua version (Lua 5.4) has switched to the indirect-threaded interpretation [30], which shows no presence of the central decode-dispatch loop. The second category of code de-virtualization work aims to remove the virtualization obfuscation layer from the lengthy execution instructions [31]–[34]; they apply dynamic taint analysis [32], enhanced backward slicing [31], [33], or improved symbolic execution [34] to extract the instructions that are related to the program semantics. Unfortunately, the nature of expensive dynamic information flow analyses (e.g., high-performance penalty and limited path coverage) has severely restricted their adoptions in production systems.

Lua's lightweight and open-source features motivate us to develop a new technique, called *interpreter semantics testing*, to reverse-engineer an obfuscated Lua bytecode file. We first pre-process the customized Lua interpreter to prepare interpreter semantics testing. In particular, we extract the obfuscated bytecode file format to generate executable bytecode.

Our key step is to find the semantics of obfuscated Lua opcodes. We modify `luac`, the Lua compiler, to precompile a set of *LuaGadgets* that can be accepted by the customized interpreter. We design each LuaGadget to test one or two specific opcodes via completing a basic Lua operation, and its result is observable. Next, we give priority to testing the LuaGadget that has the minimum opcode dependency—its opcode sequence contains the fewest unknown opcodes. We follow the above testing strategy to mutate each LuaGadget's unknown opcode values, and then the generated mutations are fed into the customized interpreter to execute. Only when the mutation hits the right opcode value, can we observe the expected LuaGadget result. We reiterate the above process until we recover the semantics of all obfuscated opcodes.

At last, we translate the obfuscated Lua bytecode file back to an executable and semantically equivalent bytecode file, which can be further processed by Lua decompiler (e.g., LuaDec [18]) and Lua code static analysis tools [18]–[20].

We have implemented our idea as an automated tool, named *LuaHunt*. LuaHunt shares the same advantage as the generic code de-virtualization methods [31]–[34]; that is, LuaHunt does not assume the specific interpreter structure in use. Moreover, as LuaGadget testing treats the bytecode's execution as a blackbox, LuaHunt can recover complete bytecode semantics but without the cost of the expensive information flow analysis. We have evaluated LuaHunt with malware samples and real-world applications that run obfuscated Lua bytecode files. For each customized interpreter, including the complicated one whose bytecode handlers are further obfuscated by code virtualization [21], [22], LuaHunt can finish the whole process of interpreter semantics testing within 90 seconds. Besides, our deobfuscation results succeed in passing the correctness testing; we utilize the Computer Language Benchmarks Game for Lua [35] to test the correctness of LuaHunt—the union of these benchmarks covers all kinds of Lua opcodes. In a nutshell, our study makes the following technical contributions:

- We study the obfuscation method adopted by an emerging IoT threat—Lua malware, which has not received much academic scrutiny. Since malware authors are capitalizing on new programming languages to better develop malware, we hope our study paints a cautionary picture for the security community on this new trend.
- Our proposed interpreter semantics testing idea represents a new direction to the efficient reverse engineering of script interpreters. LuaHunt achieves the ultimate goal of deobfuscation [36]: restoring an executable and semantically equivalent bytecode file.
- LuaHunt's performance is better than advanced code deobfuscation tools by *one or two orders of magnitude*. Security analysts utilizing LuaHunt will enjoy a simpler and more streamlined malware analysis process than ever.

We have released LuaHunt's source code and data sets to facilitate reproduction and reuse, as all found at Zenodo.

II. Background and Motivation

In this section, we first present the background information needed to understand Lua bytecode and its obfuscation. Then, we summarize the existing literature on code virtualization deobfuscation. They are the works most germane to our research, and their limitations motivate us to design LuaHunt.

A. Technical Basics of Lua Bytecode

As shown in Figure 1(b), Lua compiler (`luac`) translates a program written in the Lua programming language into a

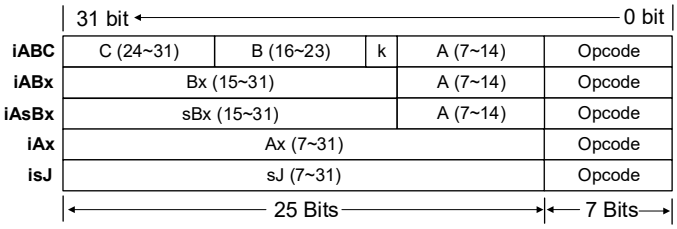

Figure 3. Lua 5.4 instruction format and five OpModes.

bytecode file that can be later loaded and executed by a Lua interpreter. Figure 2 illustrates the data structure of a Lua bytecode file. It starts with a file header (❶ in Figure 2), which stores the file's metadata information, such as the magic number, the Lua compiler's version and format, big-endian/little-endian order, and the size of different data types. Function prototype information comes after the file header, starting with a function header. The function header (❷) stores the function's metadata information, such as the source file's name, the line numbers in the source code where the function starts and stops, and the number of parameters. After the function header, the "code" field stores a list of bytecode instructions (❸).

Lua Instruction Format The format of the Lua instruction is shown in Figure 3. Each instruction is represented as an unsigned 32-bit integer, in which the opcode takes up the lower 7 bits, and the remaining bits are allocated for operands. Each opcode belongs to one of the following five *OpModes* [37]: **iABC**, **iABx**, **iAsBx**, **iAx**, and **isJ** (from top to bottom in Figure 3). They specify how to parse the higher 25 bits of operands. The number of operands varies from one to three. In the first three modes, operand A is an 8-bit unsigned integer. In iABC mode, both B and C are 8-bit integers, and there is a one-bit flag K as the signed argument. In iABx and iAsBx modes, in addition to operand A, Bx is a 17-bit unsigned number, while sBx is a 17-bit signed number. In iAx mode, Ax is a 25-bit unsigned integer. In isJ mode, sJ is a 25-bit signed integer representing jump offset, which is only used by the opcode JMP.

B. Lua Bytecode Interpreter Structures

Next, we introduce the two interpreter structures of Lua bytecode. As a scripting language's interpreter can be implemented in multiple ways [25], a general deobfuscation approach working on different interpreter structures is needed.

Decode-Dispatch Loop This is the classic way to implement a script interpreter. The virtual program counter fetches a piece of bytecode each time to decode it, and then dispatches the control flow to the opcode-associated handler function that contains the machine code to complete the calculation. Typically, the binary code of this structure has a distinctive feature in code structure: a central loop to decode, dispatch, and execute the bytecode. The virtual machines of Lua 5.1 – 5.3 are all implemented in this way.

Indirect-Threaded Code Although decode-dispatch interpretation is easy to develop, the frequently-used indirect branches caused by this structure introduce expensive mispredict penalties [38], leading to a performance slowdown. The latest version of Lua (Lua 5.4) has adopted an alternative structure, called indirect-threaded code (ITC) [30], to improve performance. The most important change of the ITC structure is removing the central decode-dispatch loop; instead, it utilizes a jump table to bridge bytecode instructions and their associated handlers. The ITC structure first establishes the jump table at compilation time to store handler function addresses; then it compiles the corresponding index of the jump table's item into the bytecode instruction.

C. Lua Bytecode Obfuscation

To stay under the detection radar, advanced Lua malware families translate the standard Lua bytecode into a new bytecode specification. Without knowledge of the new bytecode semantics, `luac` or LuaDec [18] cannot output intelligible information. As only the binary code of the interpreter is readable—but the interpreter itself does not exhibit the maliciousness of programs, security analysts have difficulty obtaining valuable insight into the malware payload.

Problem Scope Our research problem comes from the longitudinal study of real-world Lua malware and applies to other scripting languages' bytecode obfuscation as well (see "Applicability" paragraph in §VII). Our targeted Lua bytecode obfuscation still complies with Lua standard OpModes presented in Figure 3, but it involves the following modifications.

1) All fields in the file header, function header, and the opcode sequence are randomized. A customized virtual machine is attached to interpret the new bytecode file. The new bytecode can be generated at random, and thus the bytecode varies greatly from one obfuscated version to another.
2) Some non-essential file header fields (e.g., verification fields and debugging information fields) could be removed. The opcode randomization could be a one-to-one mapping or a many-to-one mapping (see "Multiple Mappings" in §IV-B).
3) As handler functions are semantically equivalent to the bytecode, to impede understanding the machine code of handler functions, malware authors can further obfuscate them by using other obfuscation schemes, such as opaque predicates [39], mixed boolean-arithmetic formulas [40], and code virtualization [21], [22].

Many obfuscation methods have been proposed to protect software in different ways, generating a large body of literature on this topic [41]–[44]. The Lua bytecode obfuscation strategies presented herein capitalize on Lua's advantages and are lightweight without adding computational overheads, but they still put reverse engineers at a disadvantage——the cost of manually analyzing an obfuscated malware sample is typically much higher than applying obfuscation. We treat a different opcode/operand encoding style as a completely different virtual instruction set architecture, and we will discuss our possible countermeasures in §VII.

IoT malware, running on resource-constrained embedded devices, has a significantly different ecosystem from traditional PC malware; any obfuscation that results in a non-negligible performance drop is highly undesirable [45]. Cozzi et al.'s

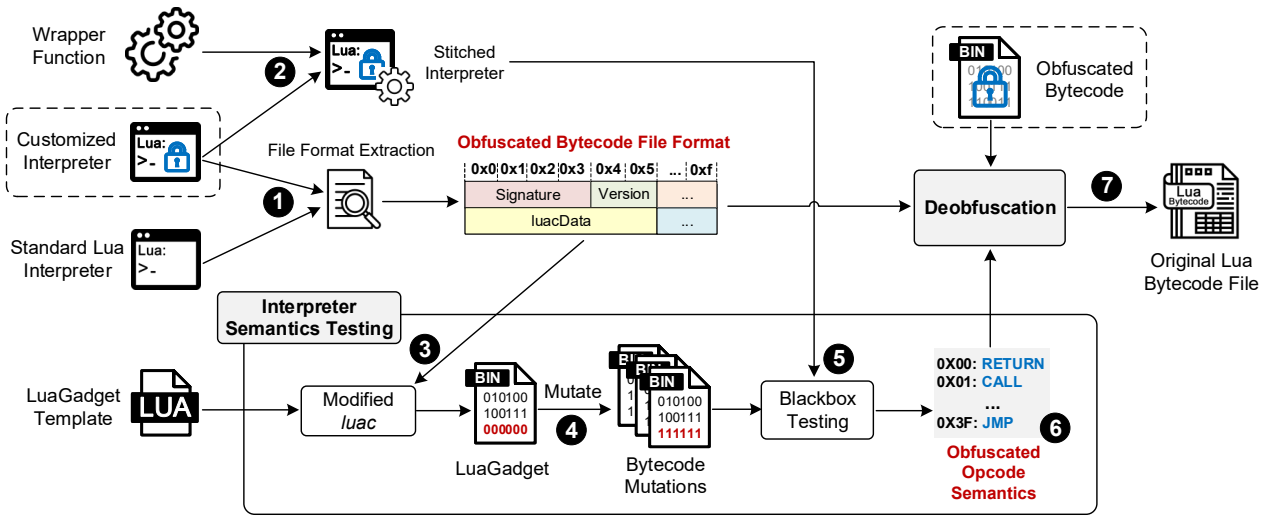

Figure 4. Overview of LuaHunt's interpreter semantics testing.

Table I
COMPARISON OF REPRESENTATIVE CODE DE-VIRTUALIZATION WORKS AND LUAHUNT

	Interpreter structures	Overhead	Deobfuscation result	Correctness testing?
Rotalumé [26]	DDL	High	Simplified CFG	No
Rolf Rolles [27]	DDL	High	Simplified CFG	No
VMAttack [29]	DDL	High	Simplified trace	No
Coogan et al. [31]	Generic	High	Simplified trace	No
Yadegari et al. [32]	Generic	High	Simplified CFG	No
BinSim [33]	Generic	High	Simplified trace	No
VMHunt [34]	Generic	High	Simplified trace	No
LuaHunt	Generic	Low	Executable program	Yes

longitudinal studies [46], [47] also confirm that, due to the concern of affecting performance and compatibility, most IoT malware samples are not obfuscated. Therefore, our targeted Lua bytecode obfuscation represents the state-of-the-art code obfuscation adopted by IoT malware.

D. Limitations of Existing Work

Lua bytecode obfuscation resembles code virtualization, which is well recognized as a highly sophisticated obfuscation technique adopted by Windows malware [48], [49]. Code virtualization converts a selected part of x86/x64 binary code into a customized bytecode format, and only the attached virtual machine can execute that bytecode. In this way, the program's original code is not present in the memory anymore. As researchers have worked on analyzing virtualization-obfuscated binaries, a natural question is whether existing approaches can be applied to Lua bytecode obfuscation.

One category of related work is limited to the classic, decode-dispatch based code virtualization [26]–[29]. They perform dynamic analysis to detect the central decode-dispatch loop and then find the mappings between executed opcodes and their associated handler functions. Next, they further simplify these handler functions and output control flow graphs (CFGs) or trace segments to understand opcodes. However, Coogan et al. [31] and VMHunt [34] confirmed that the threaded code interpretation violates the decode-dispatch loop assumption embodied by this category of work.

The recent progress in code de-virtualization is agnostic to the underlying interpreter structures [31]–[34]. These papers first record the lengthy execution traces of handler functions. Then, they remove the instructions that are not relevant to the original code semantic using an advanced, fine-grained information flow analysis, including bit-level taint-analysis [32], enhanced backward slicing for system calls [31], [33], and multiple granularity symbolic execution [34]. When applied to obfuscated Lua bytecode, they may extract opcode semantics from handler function execution traces. Unfortunately, due to the nature of expensive dynamic information flow analyses, these techniques suffer from high runtime overhead and limited opcode type coverage.

The ultimate goal of deobfuscation is "to go from obfuscated code back to the original" [36]. However, the common limitation of the above works is that they only deliver "semi-finished" products, such as simplified CFGs or trace segments. Besides, all of them lack a systematic way to test whether the deobfuscation result is correct. Table I summarizes the differences between LuaHunt and representative code de-virtualization papers, which do not fulfill our requirements. The "DDL" in Table I is short for decode-dispatch loop. As shown in the last row of Table I, our research takes full advantage of Lua's standard interpreter implementation to develop an efficient bytecode deobfuscation technique, which can produce an executable program and pass the correctness testing.

III. LUAHUNT OVERVIEW

Figure 4 illustrates the architecture of LuaHunt. The workflow of LuaHunt involves the following steps.

1. Pre-processing To generate test cases that comply with the specification of the customized interpreter, we need to figure out the layout information of various bytecode file fields and their meanings from the interpreter's binary code. The previous input format reverse engineering works have demonstrated that memory access patterns reveal the layout of the input format [50]–[53]. We first apply the mature input format extraction technique to the customized interpreter, and

then we match the layout results with standard Lua bytecode format to obtain the meanings of various bytecode file fields (❶ in Figure 4).

2. Interpreter Semantics Testing This step forms the core part of our approach—interpreter semantics testing. We first design high-quality "seed" inputs, which we call "LuaGadget Templates." Each template source file consists of a basic Lua operation (e.g., an arithmetic calculation, loading a constant, or table access) that covers one or two particular opcodes' semantics, and the operation's result is observable at run time. With the obfuscated bytecode file format obtained from the first step, we modify `luac` to compile LuaGadget Templates into LuaGadgets that can be executed by the customized interpreter (❸ in Figure 4). Specifically, we modified the bytecode file format part when generating bytecode, making it compatible with the extracted bytecode file format and ensuring that the generated LuaGadgets can be correctly loaded by the customized interpreter. We mutate LuaGadget's unknown opcode values to generate a set of bytecode files (❹) for the following blackbox testing: 1) we implement wrapper functions for different forms of customized interpreters (❷) to enable them to run specified bytecode; 2) we execute each bytecode mutation with the customized interpreter and examine the output without peering into the customized interpreter's internal structures or workings (❺). We can observe the expected LuaGadget output only when the mutation hits the exact opcode value. To find the semantics for all customized opcodes (❻) cost-efficiently, we schedule LuaGadget testing (❹ – ❺) in an order that attempts to minimize the total amount of testing.

3. Deobfuscation After we collect the results from the above steps, we translate the obfuscated Lua bytecode file format back to the standard Lua bytecode file format and rewrite the randomized bytecode instructions into the standard ones (❼ in Figure 4). Therefore, applying further code static analysis, such as decompilation [18] and bug finding [19], [20], becomes possible. Please note that, given *any* obfuscated Lua bytecode files that are compatible with that customized interpreter, we are able to reproduce their original programs rapidly without going through the above three steps again. This benefit tremendously accelerates malware analysis. We have observed that Lua malware samples take multiple obfuscated bytecode files as different malicious function modules, and a core module will schedule and control other modules. Only the executable code of the interpreter is statically visible, posing a huge challenge for malware code analysis. LuaHunt frees security professionals from the burden of manually piecing together the tedious steps of reverse engineering.

In the next section, we focus on implementation details of the key step: interpreter semantics testing.

IV. Interpreter Semantics Testing

An opcode occupies 7 bits, and thus a customized opcode could be any value between 0x00 $\sim$ 0x7F. Since the decoding mode (OpMode) of operands is decided by the opcode, as long as we find the opcode's semantics, its associated operands are also determined. LuaHunt capitalizes on blackbox testing to recover the semantics of randomized opcodes. In contrast to existing code de-virtualization works that are also agnostic to the underlying interpreter structures [31]–[34], LuaHunt's advantages are to tolerate all code obfuscation effects within handler functions and avoid analyzing tedious execution traces. The process of semantics testing is driven by the so-called "LuaGadget." Each LuaGadget is designed to test one or two specific opcodes; it is a small instruction sequence to perform a basic Lua operation and print out the result, so that we can verify the expected result at each round of testing.

```
1 LOADK     0 -1   ; Load 12345 into register 0
2 LOADK     1 -2   ; Load 56789 into register 1
3 GETTABUP  2 0 -3; Load "print" address into register 2
4 ADD       3 0 1  ; Add register 0 and register 1
5 CALL      2 2 1  ; Call "print" function to print result
6 RETURN    0 1    ; Exit Lua virtual machine
```

Figure 5. LuaGadget for testing the opcode "ADD." The corresponding source code is "print(12345+56789)."

Running Example Figure 5 shows an example of LuaGadget for testing the opcode "ADD," and "print(12345+56789)" is the original source code. Among the five opcode types, two "LOADK" instructions load addends into two registers; "ADD" is the addition operator; "GETTABUP" and "CALL" invoke a Lua built-in function to print the result to the standard output; "RETURN" means exiting Lua virtual machine. Let's first assume "ADD" is the only unknown opcode in Figure 5 (we label it as red). In that case, we mutate the "ADD" opcode from 0x00 to 0x7F and then execute each LuaGadget mutation using the customized interpreter (❺ in Figure 4). We can observe the expected result only when the mutation just hits the exact "ADD" opcode's value. In particular, we will address the following three challenges.

1) We first design LuaGadgets in Lua script form, so that we can compile them to different bytecode files that adapt to different customized interpreters. §IV-A introduces how to cover each opcode's semantics using as few as possible source code operations.
2) In Figure 5, "GETTABUP", "CALL", and "RETURN" forms an output/end module, which is necessary to a LuaGadget. Apparently, we will find their opcode values first. Besides, "ADD" depends on "LOADK" to complete the add operation. §IV-B presents our strategy to prioritize the order of LuaGadget testing.
3) In practice, the customized interpreter exists in the form of either a dynamic-link library (stand-alone ".so" file) or a static library (embedded into binary file of program). §IV-C discusses how we patch programs to run LuaGadgets using the customized interpreter.

A. LuaGadget Template Construction

We construct LuaGadgets from Lua scripts, which we call "LuaGadget Templates." We only need to develop these templates once offline and reuse them in the reverse engineering of Lua bytecode obfuscation. Every time we recompile LuaGadget Templates into a new customized bytecode form and start LuaGadget testing.

Methodology Like other high-level languages, the Lua compiler (`luac`) traverses a Lua script and generates an

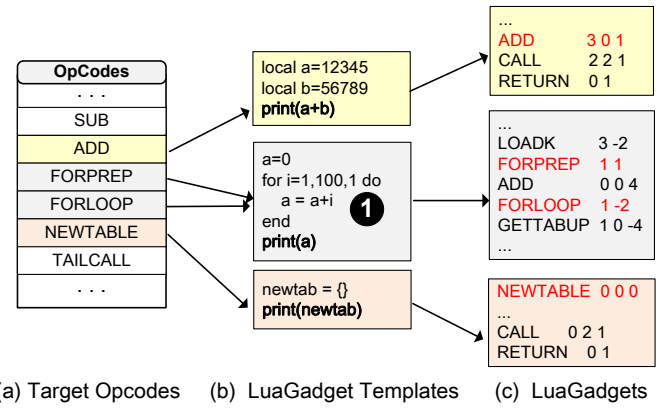

Figure 6. Examples of target opcodes and their corresponding LuaGadget Templates and LuaGadgets. Each LuaGadget is designed to test one or two specific opcodes via completing an observable calculation.

abstract syntax tree (AST), and then it translates the AST into Lua virtual machine instructions. We analyze `luac`'s source code to collect the Lua compiler's bytecode generation rules, especially how to generate opcodes according to the different operators in the Lua script. Based on the mapping between opcodes and operators in the Lua script, we design LuaGadget Templates for testing each opcode. To reduce the workload of LuaGadget testing, we hope the opcode types in each LuaGadget are as few as possible. Therefore, for each LuaGadget Template, we attempt to use as few operations as possible to complete an observable functionality involving the target opcode and then print the result. Figure 6 shows three examples of target opcodes and their corresponding LuaGadget Templates. Please note that both "FORPREP" and "FORLOOP" are always used together to form a "for" loop. For such opcodes used for control flow, we also include a simple arithmetic calculation as the loop body or branch condition (❶ in Figure 6) so that the result is observable.

Output/End Module The output/end module, which is the premise of testing other opcodes' semantics, consists of three opcodes: "GETTABUP", "CALL", and "RETURN." The challenge here is we do not want their LuaGadgets to involve other opcodes in addition to themselves. We come up with two specific LuaGadget Templates to find these three opcode values.

First, we can get a LuaGadget containing a single "RETURN" instruction by compiling an empty script file. As Lua has to exit the virtual machine after executing a bytecode file, the compiled opcode sequence is ended with a "RETURN" instruction even if the script does not have any statement. The result of running this single "RETURN" LuaGadget is also observable during testing: as long as the opcode is assigned with an incorrect value, the customized interpreter will issue an exception warning. If the execution does not lead to the exception warning, that means the opcode mutation hits the right value of "RETURN."

```
1  GETTABUP  0 0 -1; Load "print" address into register 0
2  GETTABUP  1 0 -1; Load "print" address into register 1
3  CALL      0 2 1 ; Print the address of "print"
4  RETURN    0 1   ; Exit Lua virtual machine
```

Figure 7. LuaGadget for testing GETTABUP and CALL

Table II
THE ORDER OF LUAGADGET TESTING (L0→...→L4)

Layer	LuaGadgets
L0	RETURN, (GETTABUP, CALL)
L1	RETURN0, RETURN1, SETTABUP, LOADK, LOADKX, LOADI, LOADF, LOADFALSE, LOADTRUE, LOADNIL
L4	MOVE, NEWTABLE, ADD, SUB, MUL, MOD, POW, DIV, IDIV, BAND, BOR, BXOR, BSHL, BSHR, BUNM, BNOT, NOT, ADDI, ADDK, SUBK, MULK, MODK, POWK, DIVK, IDIVK, BANDK, BORK, BXORK, SHRI, SHLI, CLOSURE, JMP, LFALSESKIP CONCAT, LEN
L3	(SETLIST, GETTABLE), GETUPVAL, SETUPVAL, GETI, SETI, GETFIELD, SETFIELD, TESTSET, TAILCALL, CLOSE (FORPREP, FORLOOP), EQ, LE, LT, TEST, MMBIN, TBC MMBINI, MMBINK, EQK, EQI, LTI, LEI, GTI, GEI, SELF, VARARG, VARARGPREP, EXTRAARG
L4	SETTABLE, TFORCALL, (TFORPREP, TFORLOOP)

Second, we generate one LuaGadget Template to cover both "GETTABUP" and "CALL." This script only has one statement: "print(print)"—calling the print function to print out its own address. Figure 7 shows the instruction sequence. If we test "RETURN" LuaGadget at first, we only need to mutate the opcode values of "GETTABUP" and "CALL" at the same time. After at most 127*126 times of trials, we can find the correct values for these two opcodes.

LuaGadget Template Compilation To generate LuaGadgets that can be executed by the customized interpreter, we modify the backend of `luac` to compile LuaGadget Templates into the bytecode that has the same file format as the obfuscated bytecode (❸ in Figure 4). All opcode values are initially set as the standard Lua opcode values. In §V-D, we will further modify `luac` to fulfill our correctness testing.

B. Test Case Prioritization

As shown in Figure 6(c), although each LuaGadget is designed to test one or two specific opcodes, its instruction sequence still contains several other opcodes to calculate the observable result and print it out. The maximum number of opcode types in our generated LuaGadgets is 11. Apparently, the cost of LuaGadget testing increases exponentially with the number of unknown opcodes. With all LuaGadgets in hand, we have to schedule their testing order to minimize the total number of mutations and testing.

According to the semantics of Lua opcodes, we classify them into the following four categories, which will help us illustrate our test case prioritization.

1) **Assignment:** opcodes in this category cover loading constants, table operations, and assigning values to registers.
2) **Basic Calculation:** basic calculation opcodes include arithmetics and bit-wise operators.
3) **Control Flow:** this kind of opcodes is used to implement various control flows, such as an unconditional jump, conditional jump, call, and loop.
4) **Language-Specific:** opcodes in this category are specific to Lua language, such as assigning vararg function arguments to registers ("VARARG") and concatenating two or more strings ("CONCAT").

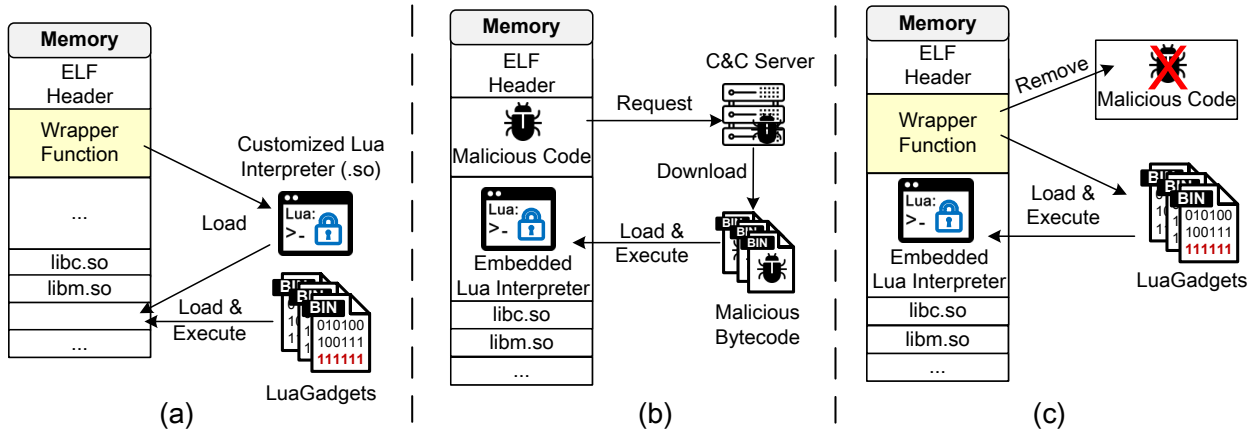

Figure 8. Running LuaGadgets with the customized interpreter. (a) The interpreter is a dynamic-link library, so we write a wrapper function to load it into memory and then execute LuaGadgets. (b) GodLua malware downloads malicious bytecode files from a C&C server and executes them. (c) We patch GodLua with the wrapper function to execute LuaGadgets.

Prioritization Strategy Our strategy to prioritize LuaGadgets is on the basis of the degree of opcode dependency. Table II shows the order of LuaGadget testing we generated, and we divide LuaGadgets into five layers. Two opcodes in parentheses mean they are tested at the same time. The different colors represent Assignment, Basic Calculation, Control Flow, and Language-Specific, respectively. The three opcodes forming the output/end module certainly have the top priority (L0 in Table II), because the remaining LuaGadgets all depend on them. After that, we traverse the remaining LuaGadgets to select the ones that can be tested on the basis of currently known opcodes. These LuaGadgets form a new layer, and their selection criterion is: the LuaGadget's opcode sequence only has one unknown opcode. For example, L1 in Table II represents the LuaGadgets having one unknown opcode based on L0; L2 represents the LuaGadgets only having one additional unknown opcode based on L0 + L1. The opcodes in L4 have complicated semantics and therefore reveal the maximum degree of opcode dependency, so we test them only when the other four layers of opcode values have been recovered to boost performance. Please note that Table II also shows four exceptions: the opcode pair in parentheses means that we have to mutate and test their values at the same time, because they are designed to be used together.

Opcode Dependency Pattern We observed some opcode dependency patterns from Table II. Most of opcodes in L1 are related to the variable assignment, such as loading a constant or boolean value into a register. Therefore, we can directly test them by printing out the variable's value. All basic calculation opcodes belong to layer L2. Due to the design of the register-based virtual machine, all arithmetics and bit-wise instructions rely on assignment opcodes to load values into registers first for calculations. Layer L3 has many opcodes used for conditional branches, because we take a simple arithmetic calculation as the loop body or branch condition to get an observable result. Language-specific opcodes have a similar dependency pattern: they also depend on assignment and calculation opcodes to deliver an observable result.

LuaGadget Mutation According to the order listed in Table II, we pass each LuaGadget to a mutation engine (❹ in Figure 4). This engine first excludes the recovered opcodes so far to limit the mutation space for the opcode to be tested. Then it mutates the unknown opcode value to generate a set of bytecode, which is later fed into the customized interpreter to find the correct opcode value.

```c
#include "lua/lua.h"
#include "lua/lualib.h"
#include "lua/lauxlib.h"

int main(int argc ,char *argv[])
{
  // Create a new Lua virtual machine state
  lua_State *L = luaL_newstate();
  // Load Lua standard library functions
  luaopen_base(L);
  // argv[1] is the name of a LuaGadget file
  luaL_dofile(L,argv[1]);
  return 0;
}
```

Figure 9. The wrapper function's source code for calling the customized Lua Interpreter.

Multiple Mappings Table II shows 83 opcodes in total; that means there are 45 reserved opcode values. Malware authors can utilize these reserved opcode values by associating them with existing opcode handlers, which results in multiple mappings between opcodes and handler functions. For example, the "MOVE" opcode may have three different values (e.g., 0x02, 0x26, 0x3F), and the mixed usage of three different "MOVE" opcode values in a bytecode file complicates reverse engineering. Therefore, when a LuaGadget execution reveals its expected result the first time, instead of terminating the testing, we will continue to mutate possible opcode values to find such a multiple mapping case.

C. Customized Interpreter Re-Stitching

Given that LuaGadget mutations are available, another problem rears its head. In most cases, the customized interpreter does not exist as a stand-alone executable file, but rather a dynamic-link library or static library. We take different measures to force the interpreter to execute LuaGadget mutations.

Dynamic-Link Library In our collected real-world Lua applications, benign programs (e.g., firmware and games) distribute the interpreter as a dynamic-link library for the ease of updating the customized interpreter. But some malware

also adopt the Lua interpreter as a dynamic-link library (e.g., LuaBot, Shishiga, Chalubo, and Remsec) for multi-platform adaptation. As shown in Figure 8(a), we write a wrapper function to call the interfaces of this dynamic-link library to execute a specified bytecode file. In particular, the wrapper function's source code is listed in Figure 9. The interfaces of Lua is declared in "lua.h", "lualib.h", and "lauxlib.h". We first initialize a new Lua virtual machine state and load Lua standard library functions (line 8 & line 10) to provide a runtime environment. Then we call "luaL_dofile" to load and execute the specified bytecode file (line 12).

Static Library Lua malware such as GodLua prefers attaching the customized interpreter as a static library. In this way, the interpreter will be embedded into the malware code to form a single file. The advantage of doing so is the interpreter and malware code can be spread together without fear of losing dynamic-link libraries. We use GodLua as an example to demonstrate how we re-stitch the Lua interpreter to apply LuaHunt. As shown in Figure 8(b), when GodLua starts running, it downloads malicious, obfuscated bytecode files from the C&C server, and then the embedded Lua interpreter will execute these downloaded bytecode files.

Figure 8(c) illustrates our method to test this kind of Lua interpreter. We patch the malware binary code with the wrapper function (Figure 9) to load and run LuaGadgets. The challenge here is the wrapper function patched into the malware binary should be address-independent like shellcode. However, the three Lua library functions called in the wrapper function (line 8, 10, and 12 in Figure 9) all depend on the fixed function addresses defined in the static library. Our solution is to use a binary diffing tool [54] to locate these three Lua library functions in the embedded Lua interpreter, and then we rewrite the wrapper function's binary code using their correct addresses. In this way, we enable the embedded, customized interpreter to execute LuaGadgets for testing.

V. Evaluation

We evaluate LuaHunt from four dimensions. First, we perform interpreter semantics testing with test cases to measure the performance of each step. We compare LuaHunt with the state-of-the-art code de-virtualization tool, VMHunt [34], to show that LuaHunt offers an complete and efficient deobfuscation solution. The second experiment evaluates a more challenging case—nested virtualization, to demonstrate the novelty of our design. Third, as few related approaches evaluate the correctness of their deobfuscation results, we design correctness testing to bridge this gap. At last, we present a case study to show that our result can assist in malware analysis.

A. Dataset and Peer Tool

Our test cases include malware samples and customized interpreters collected from an IoT device and games. They come from three different architectures: x86, MIPS (e.g., XiaoMi Router), and ARM (e.g., games). Our collected test cases represent the mainstream Lua applications in the real world.

Table III
Differences Between Customized Interpreters

Program	OF	HO	MM	VMS	Linking	Arch.	Ver.
GodLua1∼9	3	✗	✗	DDL	Static	X	5.3
GodLua10	3	✗	✓	DDL	Static	X	5.3
GodLua11∼14	3	✗	✓	ITC	Static	X	5.4
Shishiga	5	✓	✗	ITC	Dynamic	X/A/M	5.3
IoTroop	4	✗	✗	DDL	Static	M/A	5.2
Flamer	2	✗	✗	DDL	Static	X	5.1
Chalubo	2	✓	✗	DDL	Dynamic	X/A/M	5.3
Sauron	1	✓	✗	DDL	Static	X	5.2
LuaBot	5	✗	✗	DDL	Dynamic	X/A/M	5.3
Remsec	0	✗	✗	DDL	Dynamic	X	5.2
XiaoMi Router Lua	9	✗	✓	DDL	Dynamic	M	5.1
Space Hunter	3	✓	✗	DDL	Dynamic	A	5.3
Time Summon	2	✗	✗	DDL	Dynamic	A	5.3
KOF Destiny	2	✗	✓	DDL	Dynamic	A	5.2
Raziel	4	✗	✗	DDL	Dynamic	A	5.3

customized Interpreters' Diversity Table III shows the differences between interpreters in evaluation. *Obfuscated Fields (OF)* means the number of obfuscated fields in a bytecode file. *Handler Obfuscation (HO)* indicates whether additional obfuscation is applied to the opcode handlers. *Multiple Mapping (MM)* represents whether the opcode randomization uses a many-to-one mapping. *VM Structure (VMS)* shows the interpreter structure in use. DDL and ITC stand for "Decode-Dispatch Loop" and "Indirect-Threaded Code." *Linking* means whether the interpreter is embedded into the binary file via static linking or exists as a dynamic-link library. *Architecture (Arch.)* lists the supported architectures for each sample, and the "X/A/M" are short for x86, ARM, and MIPS. *Version (Ver.)* is the Lua version of each customized interpreter. To the best of our knowledge, we have not found an obfuscation tool that can achieve the diversity degree shown in Table III. We conjecture that these various obfuscation options are not from the same obfuscation tool; instead, they are implemented by different developers in an ad-hoc way. The differences between them are significant, and some of them even apply other obfuscation methods to protect opcode handlers.

Malware Samples GodLua is a representative IoT malware family applying bytecode obfuscation [7]. We used VirusTotal's Intelligence service [55] to collect 145 GodLua samples that have different hash values. We further classify these samples into 14 families, whose embedded Lua interpreters are different from each other. "GodLua1"∼"GodLua9" have a similar diversity degree. We notice GodLua samples in "GodLua10" have applied the multiple mapping in their interpreters, and "GodLua11"∼"GodLua14" have updated to Lua 5.4, which means their interpreter structures have changed from decode-dispatch loop to indirect-threaded code.

In addition to GodLua, we also obtained seven other Lua malware families from our industrial partner; they are Shishiga, IoTroop, Flamer, Chalubo, Sauron, LuaBot, and Remsec. The Shishiga family uses four different protocols (SSH, Telnet, HTTP, and BitTorrent) to implement a modular architecture. The old version of Shishiga uses the unobfuscated version of Lua script to implement its modules, but

Table IV
RUNNING TIME OF LUAHUNT

Program (# of samples)	File Format Extraction			Semantics Testing						Total	VMHunt	
	Layout	Fields	Total	L0	L1	L2	L3	L4	Total	(Col.4 + Col.10)	Performance	# of Opcodes
GodLua1∼9[1] (97)	30.1	1.3	31.4	12.6	4.3	5.9	11.7	4.4	38.9	70.3	512.7	15
GodLua10[2] (15)	30.8	1.5	32.3	12.7	5.1	6.3	13.5	4.1	41.7	74.0	N/A	N/A
GodLua11∼14[3] (33)	30.8	1.4	32.2	12.7	5.1	6.3	12.8	3.9	40.8	73.0	N/A	N/A
Shishiga (63)	28.6	1.4	30.0	16.4	5.6	7.8	13.5	3.4	46.7	76.7	653.1	16
IoTroop (51)	33.2	1.5	34.7	15.2	3.9	6.6	11.7	6.0	43.4	78.1	N/A	N/A
Flamer (19)	32.1	1.4	33.5	13.5	4.5	6.3	14.2	3.6	42.1	75.6	N/A	N/A
Chalubo (56)	28.6	1.5	29.1	13.6	5.9	4.8	10.7	3.4	38.4	67.5	N/A	N/A
Sauron (23)	25.3	1.5	26.8	15.0	4.5	4.8	14.0	4.5	42.8	69.6	N/A	N/A
LuaBot (59)	25.8	1.5	27.3	11.4	6.2	5.9	10.4	4.1	38.0	65.3	655.9	10
RemSec (40)	27.0	1.6	28.6	11.4	3.7	4.6	11.6	3.8	35.1	63.7	N/A	N/A
XiaoMi Router Lua	32.5	1.4	33.9	14.6	4.2	6.8	14.7	3.3	43.6	77.5	625.1[4]	26[4]
Space Hunter	28.3	1.0	29.3	16.3	6.1	7.5	12.9	5.2	48.0	77.3	N/A	N/A
KOF Destiny	34.3	1.2	35.5	16.2	5.9	6.7	11.5	6.9	47.2	82.7	N/A	N/A
Raziel	30.2	1.7	31.9	15.9	6.7	7.9	11.3	5.4	47.2	79.1	N/A	N/A
Time Summon	27.7	1.4	29.1	16.5	6.0	7.7	12.5	6.3	49.0	78.1	N/A	N/A

[1]We manage to obtain obfuscated Lua bytecode files for this variant.
[2]This interpreter has multiple mappings: multiple opcode values are dispatched to the same opcode handler.
[3]The customized interpreters in these four categories (GodLua11∼GodLua14) adopt the structure of Indirect-Threaded Code.
[4]As VMHunt's Pintool can only work on the x86 platform, we generate an x86-version of the customized interpreter.

we found that a new version of Shishiga had compiled Lua source code into the obfuscated bytecode format. Shishiga, IoTroop, Chalubo, and LuaBot are all IoT botnets aiming for infecting IoT devices. Flamer, Sauron, and Remsec are cyber-espionage tools used in the APT attack chain [11], [13], [14].

XiaoMi Router The second kind of our test cases comes from the customized Lua interpreter embedded in the XiaoMi router firmware. XiaoMi dominates the market share of IoT devices in China. XiaoMi develops the router firmware on top of OpenWrt, a Linux operating system targeting embedded devices. OpenWrt's configuration interface, called LuCI [56], uses Lua scripting language to implement the dynamic configuration. In the recent versions of XiaoMi router, all Lua scripts are first compiled into obfuscated bytecode files and then put into the firmware. After communicating with their developers, we know that they did this to prevent attackers from analyzing Lua scripts. XiaoMi developers admitted that, many of the previous vulnerabilities of XiaoMi router were discovered in the wild after analyzing the Lua scripts used by LuCI.

Lua Interpreters in Games An increasing number of games are using Lua as their dynamic script support [57], because Lua is very lightweight and can be easily embedded in games. Besides, the open-source framework called xLua [58] allows Unity's real-time 3D development platform [59] to support Lua programming. xLua also supports the hotfix update, so that a game can obtain the latest code updates from the server in real time without restarting. We find four games have their customized Lua interpreters to run obfuscated bytecode, including *Space Hunter* [60], *KOF Destiny* [61], *Raziel* [62], and *Time Summon* [63].

VMHunt We select VMHunt [34] to compare because it is the *only available and functional* code de-virtualization tool that is also generic to support different interpreter structures. VMHunt represents the state-of-the-art in the automated deobfuscation of malicious binary code [24]. VMHunt first records a tediously long execution trace using Intel Pin [64], and then it simplifies the virtualized code section by performing semantics-based slicing and multiple granularity symbolic execution. VMHunt's output is instruction segments that can affect program behaviors.

B. Performance Measurement

We perform all of our experiments on a consumer-grade laptop with one Intel i7-7700HQ CPU (4-Core 2.8GHz), 16GB memory, and 1TB SSD hard drive, running Ubuntu 20.04 LTS. We first construct LuaGadget Templates offline and reuse them in our evaluation. We want to iterate that LuaGadget Template construction is a one-time effort, and we can complete the construction within one hour. Table IV demonstrates the running time of LuaHunt. Column 2–11 of Table IV shows the running time of LuaHunt's online steps when performing interpreter semantics testing with our test cases. For each malware family, the number in parentheses means the number of malware samples. The "Layout" and "Fields" columns represent the running time of layout extraction and fields matching. L0∼L4 indicate the steps of LuaGadget testing defined in Table II. "VMHunt" columns show the average running time (seconds) and the number of recovered opcodes by one time of VMHunt's trace-based analysis. Our file format extraction can be finished in 40 seconds. After that, the semantics testing takes less than 50 seconds. Among the five layers of LuaGadget testing, we find that testing L0 and L3 takes the longest time. The reason is that they have pairs of opcodes that need to be mutated and tested at the same time: L0 has a pair of such opcodes, and L3 has two pairs. However, the running time of testing L3 is not much longer than that of L0. This is because when testing L3, we have removed known opcode values to limit the mutation scope. After semantics testing, given an obfuscated bytecode file, we only need to take an additional one second to rewrite

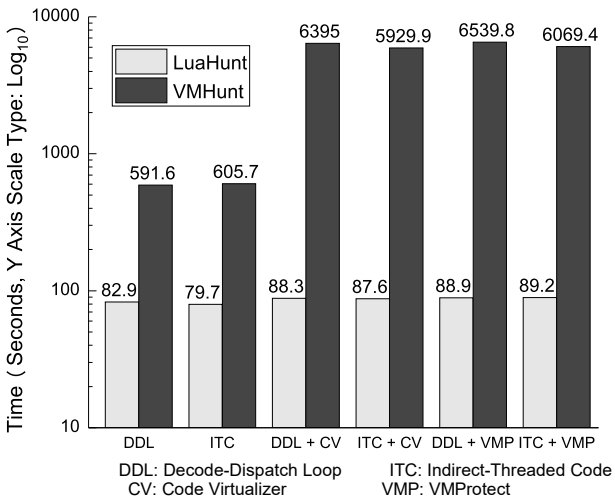

Figure 10. Running time (seconds) of LuaHunt and VMHunt when handling nested virtualization.

it back to a standard Lua bytecode file. Overall, the whole process of deobfuscation can be completed within 90 seconds.

A common observation from Table IV is that LuaHunt's running time does not fluctuate much across different interpreters. Because for every interpreter, we go through a similar LuaGadget testing process to recover all randomized opcode values. It is the number of opcodes to be tested decides the upper limit of LuaHunt's overhead. Note that in Table IV, we only obtain obfuscated Lua bytecode files for four cases: GodLua5, Shishiga, LuaBot, and XiaoMi Router; but LuaHunt can still work in the presence of only the customized interpreter—we generate LuaGadgets to force the execution of the interpreter. LuaHunt outperforms VMHunt with regard to full opcode type coverage and low overhead. As shown in the last two columns of Table IV, VMHunt only succeeded in the four cases whose obfuscated Lua bytecode files are available. Due to the high overhead caused by VMHunt's multiple granularity symbolic execution, VMHunt suffers at least 7X performance slowdown, and its overhead upper limit is decided by the size of the execution trace. Besides, VMHunt's performance data only reflects one time of execution trace analysis, which means VMHunt only recovers a limited number of opcodes that appeared in a single trace, such as 15 for GodLua5, 16 for Shishiga, and 10 for LuaBot.

C. Nested Virtualization

The VMHunt paper also evaluated a more challenging obfuscation case that applied another layer of code virtualization to a virtual instruction's native handler function, which is the so-called "nested virtualization." VMHunt is able to deal with this complicated case layer-by-layer at the cost of much larger runtime overhead. In contrast, as LuaHunt takes the opcode handler functions as a blackbox, applying further obfuscation on handler functions has little impact on LuaHunt's performance. We select the benchmark SHA-1 from Table V as input. We first generate two obfuscated Lua bytecode versions with different interpreter structures: decode-dispatch loop (DDL) and indirect-threaded code (ITC). Then, for each interpreter structure, we apply Code Virtualizer [21] and VMProtect [22] to further obfuscate opcode handler functions, respectively.

As shown in Figure 10, we get six bytecode files with different obfuscation/virtualization combinations. LuaHunt's running time remains stable in all six cases, and we attribute the small performance degradation in the last four cases to the two layers of virtual machine execution. However, VMHunt's cost when analyzing the four nested virtualization samples stands in stark contrast to that of LuaHunt—about 67X∼73X slowdown. The root cause is after applying code virtualization, the trace size has five orders of magnitude explosion.

We stress that our comparison to VMHunt is not a criticism of its technique, but rather offers a new idea of reverse-engineering script interpreters efficiently. LuaHunt relies on the construction of executable LuaGadgets to drive blackbox testing iterations, while VMHunt requires less knowledge on the interpreter; it only performs trace-based data flow analysis. We will further discuss this tradeoff in §VII. We believe the benefits of using LuaHunt far outweigh the drawbacks.

D. Correctness Testing

All of our test cases are either malware or real-world applications; for most of them, we cannot even get their bytecode files. For example, all C&C hosts of our collected GodLua samples have been taken down, so we are unable to download their malicious bytecode files. The lack of ground truth (e.g., the original bytecode file before obfuscation) casts doubt on how to demonstrate our deobfuscation result is correct. In this paper, we define the "correctness" as follows: whether we can recover obfuscated bytecode file format and find the right values for *all* obfuscated opcodes.

To bridge this gap, we further modify `luac` to design the correctness testing. First, after opcode deobfuscation, we feed the right opcode values back to `luac`. In this way, given any Lua script, we can independently compile it into the bytecode that can be smoothly executed by the customized interpreter. Second, we select the Computer Language Benchmarks Game for Lua [35] to test the correctness of LuaHunt. These benchmark programs are initially developed to evaluate the efficiency of Lua language. However, we find that the union of their bytecode covers all kinds of Lua opcodes. The description of each benchmark and the number of opcodes covered are shown in Table V. Therefore, for each test case, we compile benchmark programs into customized bytecode files and run them using the customized interpreter. As benchmark programs have provided a test suite, if we got it wrong on any input field format or opcode semantics, the test suite executions cannot be all successful. We reiterate this correctness testing on all customized interpreters in Table IV and none failed.

We conduct a separate experiment to demonstrate how a failed opcode value recovery could affect the correctness evaluation result. We set eight opcode values incorrectly on purpose and test how many benchmarks will fail accordingly at runtime. Table VI shows the validation result. This table shows that if an opcode was wrong, which benchmark's execution does not pass the correctness testing. The abbreviations of the

Table V
THE COMPUTER LANGUAGE BENCHMARKS GAME FOR LUA

Benchmarks	Description	#Opcodes
Binary Trees	This program creates perfect binary trees using minimum number of allocations.	18
Fannkuch Redux	The fannkuch benchmark comes from the paper "Performing Lisp Analysis of the FANNKUCH Benchmark" [65].	20
Fasta	This program takes two actions: 1) generate DNA sequences by copying from a given sequence; 2) generate DNA sequences by weighted random selection from 2 alphabets.	27
K-nucleotide	Mapping the DNA letters to the bytes 0, 1, 2, 3, and using a hash function to concatenate those two-byte codes.	25
Mandelbrot	Calculating Mandelbrot Set [66].	17
N Body	Modelling the orbits of Jovian planets with the same simple symplectic-integrator.	21
Reverse Complement	Reverse Complement converts a DNA sequence into its reverse, complement, or reverse-complement counterpart.	20
Spectral Norm	Solving the Hundred-Dollar, Hundred-Digit Challenge Problems [67].	18
SHA-1	A 160-bit Secure Hash Algorithm.	27

Table VI
CORRECTNESS VALIDATION RESULT

Opcode	BT	FR	FT	KN	MB	NB	RC	SN	SHA-1
GETTABUP	✗	✗	✗	✗	✗	✗	✗	✗	✗
LOADBOOL	✓	✗	✓	✗	✓	✓	✓	✓	✓
DIV	✓	✓	✗	✗	✗	✗	✓	✗	✗
POW	✗	✓	✓	✓	✓	✓	✓	✓	✓
JMP	✗	✗	✗	✗	✗	✗	✗	✗	✗
FORLOOP	✗	✗	✗	✗	✗	✗	✗	✗	✗
CONCAT	✓	✓	✗	✓	✓	✓	✗	✓	✗
LEN	✓	✓	✗	✓	✓	✗	✓	✓	✗

first row are "Binary Trees," "Fannkuch Redux," "Fasta," "K-nucleotide," "Mandelbrot," "N Body," "Reverse Complement," and "Spectral Norm." As all benchmarks rely on GETTABUP, JMP, and FORLOOP, their wrong opcode values make all benchmark executions crash. On the other end of the spectrum, only the "Binary Trees" benchmark uses the exponentiation operator POW, and hence only one benchmark fails when POW's value is incorrect. The union of our selected benchmarks ensure that we do not miss any failed bytecode reverse engineering steps.

For XiaoMi router's firmware, as we can extract the standard Lua scripts from its old version (version 2.19.40), we also compare them with our deobfuscation results. In particular, we first deobfuscate the obfuscated Lua bytecode in the latest version of XiaoMi router firmware (version 2.22.19). Then, we use LuaDec [18] to decompile LuaHunt's result back to Lua scripts. Since all debugging symbols have been removed from the obfuscated bytecode, LuaDec's output is unable to show intelligible variable names. Nevertheless, our manual comparison confirms that the control flow relations are equivalent. Besides, we put deobfuscated bytecode files in the old firmware version to replace the original Lua scripts, and XiaoMi router can still work properly, which also indicates that our deobfuscation result is correct.

E. Lua Malware Analysis Case Study: GodLua

As the code size of "Upgrade.png" in Figure 11 is small, we list the decompilation result of its bytecode file based on LuaHunt's output in Figure 12. LuaHunt can work in the presence of only the customized interpreter. For different obfuscated bytecode files sharing the same customized interpreter, we only need to perform LuaHunt's interpreter semantics testing once; LuaHunt's result is reused to translate an obfuscated bytecode file into a standard Lua bytecode file within one second.

Although all C&C hosts of our collected GodLua samples have been taken down, we manage to extract four malicious

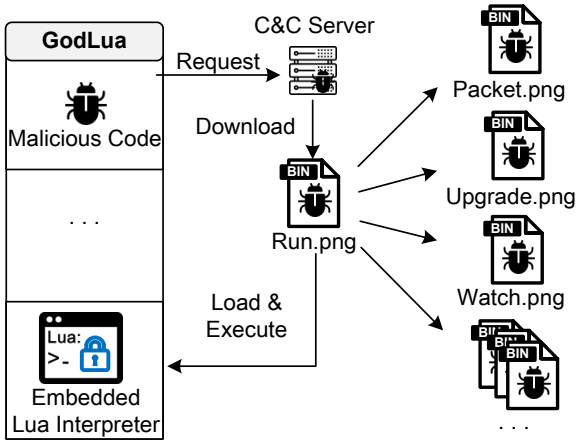

Figure 11. Reverse-engineering GodLua bytecode files.

```
1 local l_0_0 = (_Env.require)("common.util")
2 local l_0_1 = {}
3 l_0_1.handle = function(l_1_0)
4   -- function num : 0_0 , upvalues : _Env, l_0_0
5   if not l_1_0 then
6     return (_Env.Env).Version
7   end
8   if (_Env.Env).System == "Linux" and (_Env.Env).Version <
      l_1_0 then
9     (l_0_0.system)("rm -rf " .. (_Env.Env).File)
10    ;
11    (l_0_0.download)("https://d.cloudappconfig.com/" .. (
      _Env.Env).Cross .. "/Satan", (_Env.Env).File)
12    ;
13    (l_0_0.system)("chmod 777 " .. (_Env.Env).File)
14    ;
15    (l_0_0.system)("cat /dev/shm/.p | xargs kill;" .. (_Env
      .Env).File)
16  end
17  return (_Env.Env).Version
18 end
19
20 %return l_0_1
21 %
```

Figure 12. Decompilation result of one GodLua bytecode file ("Upgrade.png" in Figure 11) based on LuaHunt's output.

bytecode files from VirusTotal's PCAP files.[1] All of these four bytecode files disguise their filename extensions as ".png". We deobfuscate them using 14 different opcode sequences obtained from Table IV, and eventually, we find that they all use the same customized interpreter as "GodLua5." Based on LuaHunt's output, we are able to demystify the correlation between these four malicious bytecode files.

As shown in Figure 11, GodLua adopts modular programming. "Run.png" is the core of GodLua, scheduling other modules to communicate with the C&C server and perform HTTP flood attacks. "Run.png" can process eight C&C commands and dispatch them to other modules. These eight C&C commands include HANDSHAKE (0x01), HEARTBEAT (0x02), LUA (0x03), SHELL (0x04), UPGRADE (0x05), QUIT (0x06), SHELL2 (0x07), and PROXY (0x08).

"Packet.png" is the module responsible for communicating with the C&C server using socket; it is called by "Run.png" when communicating with the C&C server, such as sending HEARTBEAT messages and receiving C&C commands. "Upgrade.png" is the module for updating GodLua's bytecode; it downloads the latest bytecode from the C&C server, adds execution permissions to the bytecode, kills the corresponding old process, and then executes the new bytecode. "Watch.png" keeps GodLua alive in the target operating system. As the GodLua process could be killed for some reason, "Watch.png" will create a scheduled task for the system and periodically detect whether GodLua is still alive. These modules work together to prolong GodLua's lifetime and execute attack commands issued by the C&C server.

VI. Related Work

Scripting Language Deobfuscation Existing deobfuscation methods in this direction are orthogonal to LuaHunt; they focus on reverse-engineering obfuscated source code [69]–[72], rather than obfuscated bytecode. JSDES [70] conducts dynamic analysis to log the possible functions that are used to run the obfuscated JavaScript code. Li et al.'s work [71] performs an emulation-based recovery at the level of subtrees in the abstract syntax tree of PowerShell scripts. Lu et al. [69] collect JavaScript bytecode execution traces and apply backward slicing for system calls to obtain semantically relevant instructions, which are further translated to JavaScript source code. Script-level obfuscation (e.g., code encryption and variable renaming) can be defeated by analyzing bytecode execution traces, because opcodes do not change. In contrast, the problem we solved is more challenging, because the bytecode is not analyzable after opcode obfuscation, and the original program's bytecode is never restored anywhere.

Lua Code Security Analysis Compared with other increasingly popular programming languages, security analysis research for Lua code is lacking. Andrei Costin [19] developed the first static analysis tool to find vulnerable Lua script code. BMCLua [20] performs formal verification to detect Lua script bugs via SMT-based bounded model checking [73]. Park et al. [74] demonstrate that Lua interpreters are vulnerable to bytecode corruption attacks, which can lead to arbitrary code execution. Exploiting vulnerabilities within malicious code can potentially help cyber-defenders terminate malware execution or take down botnets [75]. As LuaDec [18] can decompile LuaHunt's output back to malware source code for further security analysis, LuaHunt is an appealing complement to existing Lua script bug-finding tools [19], [20].

[1]PCAP (Packet Capture) files contain malware network packet data when executed in a sandbox environment [68].

VII. Discussion

The cyber arms race between malware analysis and its countermeasures has transformed into an intensive tug-of-war. A natural question is whether script-based malware authors can easily impede LuaHunt once it is publicly known. We do not assume that evading our approach is strictly impossible, but it can prohibitively increase adversaries' costs. This section discusses LuaHunt's applicability and adversary analysis.

Applicability Other scripting languages' bytecode obfuscation [76]–[80] closely resembles Lua bytecode obfuscation. For example, the client of Dropbox, a very popular file hosting service, consists of a modified Python interpreter running obfuscated Python bytecode [76], and its opcodes are also randomized. Security analysts found that new Python malware whose bytecode files are obfuscated in the same way as GodLua [78], [80]. JavaScript Bytecode Compiler [79] and Quarkslab [77] obfuscate JavaScript/Python bytecode by randomizing opcodes, which are used together with the provided customized virtual machine. Although this paper focuses on Lua bytecode deobfuscation, we stress that the methodology of our interpreter semantics testing (e.g., LuaGadget Template construction and LuaGadget prioritization) is general and applicable to other scripting languages as well.

Anti-LuaGadget-Testing As discussed in §II-C, current Lua bytecode obfuscation represents a good tradeoff between obfuscation strength, runtime performance, and development cost. We rely on the semantics of the standard Lua bytecode instruction set to construct LuaGadgets for semantics testing. To deter the generation of LuaGadgets, adversaries can redesign a new virtual instruction set architecture (ISA) that is totally different from the standard Lua ISA in opcode size and operand decoding mode. The fundamental question here is how much customization malware authors can perform on a Lua interpreter for it to become a new interpreter for a new language, just like "The Ship of Theseus."

Our countermeasure is to first recover the new ISA details and then apply LuaHunt. In particular, we will first perform dynamic input format extraction [50]–[52] to recover the new virtual ISA's format from execution traces and then adopt LuaHunt to deobfuscate bytecode. We did an experiment to estimate the impact on our performance via dynamically recovering the new ISA. We treat Lua's ISA as an unknown ISA and adopt Tupni [51]'s technique to reverse-engineer the ISA format. Tupni's analysis takes about 7 minutes, plus LuaHunt's 1.5 minutes running time; the overall performance is still better than performance-heavy information flow analysis methods like VMHunt. However, having this new ISA also abandons Lua's unique advantages, and it will push adversaries to redevelop a new scripting language compiler and interpreter, which is a non-trivial task for skilled malware developers.

VIII. Conclusion

Cybercriminals are using Lua as a new programming language to better develop IoT malware. Furthermore, they obfuscate malicious Lua bytecode files to frustrate malware analysis. In this paper, we first demystify the inner mechanism of Lua bytecode obfuscation. Then, we develop and evaluate LuaHunt, an efficient technique to reverse-engineer obfuscated Lua bytecode via the novel interpreter semantics testing idea, which is free from analyzing the tediously-long execution traces of opcode handlers. LuaHunt reveals a very small runtime overhead and significantly reduces the workload of security analysts; it represents a new direction to the efficient reverse engineering of bytecode obfuscation in interpreters of script languages.

References

[1] PYPL Index. PYPL PopularitY of Programming Language. http://pypl.github.io/PYPL.html, [online].

[2] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. A Look at the Design of Lua. *Communications of the ACM*, 61(11), 2018.

[3] Wikipedia. List of applications using Lua. https://en.wikipedia.org/wiki/List_of_applications_using_Lua, [online].

[4] Pierluigi Paganini. LuaBot is the First Linux DDoS Botnet Written in Lua Language. https://securityaffairs.co/wordpress/51155/malware/linux-luabot.html, September 2016.

[5] x0rz. Interview with the LuaBot Malware Author. https://medium.com/@x0rz/interview-with-the-luabot-malware-author-731b0646fc8f, September 2016.

[6] Michal Malik. Linux Shishiga malware using LUA scripts. https://www.welivesecurity.com/2017/04/25/linux-shishiga-malware-using-lua-scripts/, April 2017.

[7] Sergiu Gatlan. New GodLua Malware Evades Traffic Monitoring via DNS over HTTPS. http://tiny.cc/k198tz, July 2019.

[8] GReAT. ProjectSauron: Top Level Cyber-Espionage Platform Covertly Extracts Encrypted Government Comms. https://securelist.com/faq-the-projectsauron-apt/75533/, August 2016.

[9] Warren Mercer, Paul Rascagneres, and Vitor Ventura. PoetRAT: Malware targeting public and private sector in Azerbaijan evolves. https://blog.talosintelligence.com/2020/10/poetrat-update.html, October 2020.

[10] Check Point Research. IoTroop Botnet: The Full Investigation. https://research.checkpoint.com/2017/iotroop-botnet-full-investigation/, October 2017.

[11] Darien Kindlund. Flamer/sKyWIper Malware: Analysis. https://www.fireeye.com/blog/threat-research/2012/05/flamerskywiper-analysis.html, May 2012.

[12] Binary Defense. Chalubo Botnet. https://www.binarydefense.com/threat_watch/chalubo-botnet/, October 2018.

[13] Nicholas Weaver. What Sauron Tells Us About What NSA's Up To, and What It Should Do Next. http://tiny.cc/2d6ytz, August 2016.

[14] Warwick Ashford. Strider cyber attack group deploying malware for espionage. http://tiny.cc/9d6ytz, August 2016.

[15] StackHawk. Lua XSS: Examples and Prevention. https://www.stackhawk.com/blog/lua-xss-examples-and-prevention/, March 2022.

[16] BlackBerry Research. Threat Thursday: SunSeed Malware Targets Ukraine Refugee Aid Efforts. https://blogs.blackberry.com/en/2022/03/threat-thursday-sunseed-malware, March 2022.

[17] 360 Network Security Research Lab. An Analysis of GodLua Backdoor. https://blog.netlab.360.com/an-analysis-of-godlua-backdoor-en/, July 2019.

[18] Hisham Muhammad and Zsolt Sztupak. LuaDec is a Decompiler for the Lua Language. https://github.com/viruscamp/luadec, [online].

[19] Andrei Costin. Lua Code: Security Overview and Practical Approaches to Static Analysis. In *Proceedings of the 2017 IEEE Security and Privacy Workshops*, 2017.

[20] Felipe R. Monteiro, Francisco A.P. Januário, Lucas C. Cordeiro, and Eddie B. de Lima Filho. BMCLua: A Translator for Model Checking Lua Programs. *ACM SIGSOFT Software Engineering Notes*, 42(3), 2017.

[21] Oreans Technologies. Code Virtualizer: Total Obfuscation against Reverse Engineering. http://oreans.com/codevirtualizer.php, [online].

[22] VMProtect Software. VMProtect software protection. http://vmpsoft.com, [online].

[23] Lei Xue, Yuxiao Yan, Luyi Yan, Muhui Jiang, Xiapu Luo, Dinghao Wu, and Yajin Zhou. Parema: An Unpacking Framework for Demystifying VM-Based Android Packers. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*, 2021.

[24] Shijia Li, Chunfu Jia, Pengda Qiu, Qiyuan Chen, Jiang Ming, and Debin Gao. Chosen-Instruction Attack Against Commercial Code Virtualization Obfuscators. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS'22)*, 2022.

[25] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[26] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*, 2009.

[27] Rolf Rolles. Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT'09)*, 2009.

[28] Yoann Guillot and Alexandre Gazet. Automatic Binary Deobfuscation. *Journal in Computer Virology*, 6(3), 2010.

[29] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES'17)*, 2017.

[30] Robert B. K. Dewar. Indirect Threaded Code. *Communications of the ACM*, 18(6), 1975.

[31] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[32] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[33] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*, 2017.

[34] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. VMHunt: A Verifiable Approach to Partial-Virtualized Binary Code Simplification. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, page 442–458, 2018.

[35] Isaac Gouy. The Computer Benchmark Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/measurements/lua.html, [online].

[36] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic Deobfuscation: From Virtualized Code Back to the Original. In *Proceedings of the 15th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'18)*, 2018.

[37] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua 5.4 Reference Manual. https://www.lua.org/manual/5.4/manual.html, 2020.

[38] M. Anton Ertl and David Gregg. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. In *Proceedings of the 2001 European Conference on Parallel Processing*, 2001.

[39] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, 1998.

[40] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. Boosting SMT Solver Performance on Mixed-Bitwise-Arithmetic Expressions. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'21)*, 2021.

[41] Jasvir Nagra and Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[42] Kevin A. Roundy and Barton P. Miller. Binary-code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, 46(1), 2013.

[43] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Computing Surveys*, 49(1), April 2016.

[44] Sebastian Banescu and Alexander Pretschner. *Advances in Computers*, volume 108, chapter Chapter Five - A Tutorial on Software Obfuscation. Elsevier, 2018.

[45] Jinchun Choi, Afsah Anwar, Hisham Alasmary, Jeffrey Spaulding, DaeHun Nyang, and Aziz Mohaisen. IoT Malware Ecosystem in the Wild: A Glimpse into Analysis and Exposures. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019.

[46] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding Linux Malware. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P'18)*, 2018.

[47] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. The Tangled Genealogy of IoT Malware. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC'20)*, 2020.

[48] Michalis Polychronakis. *Reverse Engineering of Malware Emulators*, chapter Encyclopedia of Cryptography and Security. Springer US, 2011.

[49] Ramya Manikyam, J. Todd McDonald, William R. Mahoney, Todd R. Andel, and Samuel H. Russ. Comparing the Effectiveness of Commercial Obfuscators Against MATE Attacks. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW'16)*, 2016.

[50] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[51] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, 2008.

[52] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. PatchScope: Memory Object Centric Patch Diffing. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS'20)*, 2020.

[53] Zhiqiang Lin and Xiangyu Zhang. Deriving Input Syntactic Structure from Execution. In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE '08)*, 2008.

[54] zynamics. BinDiff: Graph Comparison for Binary Files. https://www.zynamics.com/bindiff.html, 2020.

[55] VirusTotal. VT Intelligence: Combine Google and Facebook and apply it to the field of Malware. https://www.virustotal.com/gui/intelligence-overview, [online].

[56] OpenWrt Project. LuCI - OpenWrt Configuration Interface. https://github.com/openwrt/luci, [online].

[57] Mehedi Hasan. Best Programming Language for Games: 15 Game Programming Languages Reviewed. https://gamedev.stackexchange.com/questions/11/what-scripting-language-should-i-choose-for-my-game, 2020.

[58] Tencent. xlua. https://github.com/Tencent/xLua/, [online].

[59] Unity Technologies. Unity Real-Time Development Platform. https://unity.com/, [online].

[60] YinHan Games. Space Hunter. http://hunter.yh.cn/hunter/index, [online].

[61] Tencent. King of Fighters. https://kofd.qq.com/main.shtml, [online].

[62] Tencent. Raziel. https://raz.qq.com/, [online].

[63] YinHan Games. Time Summon. https://moba.yh.cn/moba/v2/pc/index.html, [online].

[64] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, 2005.

[65] Kenneth R Anderson and Duane Rettig. Performing Lisp Analysis of the FANNKUCH Benchmark. *ACM SIGPLAN Lisp Pointers*, 7(4):2–12, 1994.

[66] Wolfram Math World. Mandelbrot Set. https://mathworld.wolfram.com/MandelbrotSet.html, [online].

[67] Wolfram Math World. Hundred-Dollar, Hundred-Digit Challenge Problems. https://mathworld.wolfram.com/Hundred-DollarHundred-DigitChallengeProblems.html, [online].

[68] Emiliano Martinez. VirusTotal += PCAP Analyzer. https://blog.virustotal.com/2013/04/virustotal-pcap-analyzer.html, April 2013.

[69] Gen Lu and Saumya Debray. Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach. In *Proceedings of the IEEE 6th International Conference on Software Security and Reliability (SERE'12)*, SERE'12, 2012.

[70] Moataz AbdelKhalek and Ahmed Shosha. JSDES: An Automated De-Obfuscation System for Malicious JavaScript. In *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES'17)*, 2017.

[71] Zhenyuan Li, Qi Alfred Chen, Chunlin Xiong, Yan Chen, Tiantian Zhu, and Hai Yang. Effective and Light-Weight Deobfuscation and Semantic-Aware Attack Detection for PowerShell Scripts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*, 2019.

[72] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. Stochastic Optimization of Program Obfuscation. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*, 2017.

[73] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-Based Bounded Model Checking for Embedded ANSI-C Software. *IEEE Transactions on Software Engineering*, July 2012.

[74] Taemin Park, Julian Lettner, Yeoul Na, Stijn Volckaert, and Michael Franz. Bytecode Corruption Attacks Are Real—And How to Defend Against Them. In *Proceedings of the 2018 International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'18)*, 2018.

[75] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babić, and Dawn Song. Input Generation via Decomposition and Re-Stitching: Finding Bugs in Malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[76] Dhiru Kholia and Przemysław Węgrzyn. Looking inside the (Drop) Box. In *Proceedings of the 7th USENIX Conference on Offensive Technologies (WOOT'13)*, 2013.

[77] Serge Guelton. Building an Obfuscated Python Interpreter: we need more opcodes. https://bit.ly/2Y9mIdn, May 2014.

[78] Joshua Homan. Deobfuscating Python Bytecode. https://www.mandiant.com/resources/deobfuscating-python, May 2016.

[79] Johannes Willbold. Rusty-JSYC (JavaScript bYtecode Compiler) is a JavaScript-To-Bytecode Compiler Written in Rust. https://github.com/jwillbold/rusty-jsyc, 2019.

[80] Austin Jackson. Python Malware on the Rise. https://www.cyborgsecurity.com/cyborg_labs/python-malware-on-the-rise/, July 2020.

Chenke Luo is currently pursuing his Ph.D. in the School of Cyber Science and Engineering at Wuhan University under the supervision of Dr. Jianming Fu. His current research focuses on system security and software security.

Jiang Ming is an Assistant Professor in the Department of Computer Science at Tulane University. He received his PhD from The Pennsylvania State University. His research interests span Software and Systems Security. He strives to ground his efforts in practical security problems with an eye towards developing effective solutions to address realistic threats caused by today's emerging technologies.

Jianming Fu received his Ph.D. degree from Wuhan University, Wuhan, China, in 2000. He is currently a professor at the School of Cyber Science and Engineering, Wuhan University. His research interests include system security, software security, AI security, and mobile security.

Guojun Peng received his Ph.D. degree from Wuhan University, Wuhan, China, in 2008. He is currently a professor at the School of Cyber Science and Engineering, Wuhan University. His research interests include malware analysis and defense, software security, mobile security, and trusted computing.

Zhetao Li (M'17) is a professor in College of Information Science and Technology, Jinan University. He received the B.Eng. degree in Electrical Information Engineering from Xiangtan University in 2002, the M.Eng. degree in Pattern Recognition and Intelligent System from Beihang University in 2005, and the Ph.D. degree in Computer Application Technology from Hunan University in 2010. He is a member of IEEE and CCF.