2017 IEEE Symposium on Security and Privacy Workshops

Lua code: security overview and practical approaches to static analysis

Andrei Costin*†

*University of Jyväskylä
Jyväskylä, Finland
ancostin@jyu.fi

†Firmware.RE
andrei@firmware.re

***Abstract*—Lua is an interpreted, cross-platform, embeddable, performant and low-footprint language. Lua's popularity is on the rise in the last couple of years. Simple design and efficient usage of resources combined with its performance make it attractive for production web applications even to big organizations such as Wikipedia, CloudFlare and GitHub. In addition to this, Lua is one of the preferred choices for programming embedded and IoT devices. This context allows to assume a large and growing Lua codebase yet to be assessed. This growing Lua codebase could be potentially driving production servers and extremely large number of devices, some perhaps with mission-critical function for example in automotive or home-automation domains.**

However, there is a substantial and obvious lack of static analysis tools and vulnerable code corpora for Lua as compared to other increasingly popular languages, such as PHP, Python and JavaScript. Even the state-of-the-art commercial tools that support dozens of languages and technologies actually do not support Lua static code analysis.

In this paper we present the first public Static Analysis for Security Testing (SAST) tool for Lua code that is currently focused on web vulnerabilities. We show its potential with good and promising preliminary results that we obtained on simple and intentionally vulnerable Lua code samples that we synthesized for our experiments. We also present and release our synthesized corpus of intentionally vulnerable Lua code, as well as the testing setups used in our experiments in form of virtual and completely reproducible environments. We hope our work can spark additional and renewed interest in this apparently overlooked area of language security and static analysis, as well as motivate community's contribution to these open-source projects. The tool, the samples and the testing VM setups will be released and updated at http://lua.re and http://lua.rocks.

I. Introduction

Lua is an interpreted, cross-platform, embeddable, performant and low-footprint language that supports *"extensible semantics, anonymous functions, full lexical scoping, proper tail calls, and coroutines"* [1]. While not yet extremely visible in the overall programming languages ecosystem, Lua's popularity is on the rise in the last couple of years. For example, in January 2017 it ranks above languages such as Transact-SQL [1] and Scala according to TIOBE index of programming languages' popularity [2], and above languages such as Go and Delphi according to PYPL index [3]. On the one hand, Lua's lack of sufficient visibility could be explained by a series of factors, such as quantitative and qualitative lack of libraries in LuaRocks compared to RubyGems (for Ruby), PyPI (for Python) and CPAN (for Perl) [4]. On the other hand, Lua's rise in popularity [2], [5] could be explained by the following aspects. First, the simple design combined with efficient performance and resource usage [6], [7] make it attractive for production web applications even to big organizations such as Wikipedia [8], CloudFlare [9] and GitHub [10]. Second, there is an exploding market and an overwhelming popularity of extremely low-cost IoT chipsets such as ESP8266 [11] that can run Lua code using open-source firmware such as NodeMCU [12] and NodeLua [13]. Additionally, some of devices running Lua could be used in mission-critical applications, such as automotive [2], radiation/photon experimental physics [3], home automation and more [14].

Within the described context, there are several main motivations for our work. First, there is a substantial lack of static analysis tools for Lua as compared to other increasingly popular languages, such as PHP, Python, and JavaScript [15], [16]. In fact, we are only aware of very few such tools that perform static analysis on Lua code. *LuaCheck* [17], *lualint* [18] and *lua-checker* [19] are mainly intended for linting and checking the code style rather than performing static code analysis and looking for most common security issues (e.g., *OWASP Top10* [20]), such as XSS and SQLi. Second, even the state-of-the-art commercial tools that support dozens of languages and technologies actually do not support Lua applications [21], [22], [23]. Even when they do [24], those tools unfortunately support Lua only in dynamic blackbox analysis mode [25], i.e., static analysis of Lua code is not possible. Finally, there is a

[1]Transact-SQL ranked as most popular language in TIOBE index 2013 [2].

[2]See Mercedes-Benz FOSS licenses used within each car category http://www4.mercedes-benz.com/manual-cars/ba/foss/content/en/assets/FOSS_licences.pdf

[3]See APS ANL EPICS Attocube ANC300 Piezo Motion Controller usage https://www1.aps.anl.gov/files/download/SECTOR27/Manual%20ANC300%20v3.1.pdf

DOI 10.1109/SPW.2017.38

lack of knowledge on insecure coding patterns and insecure API calls for Lua code. Baseline corpora of vulnerable samples and applications exist for other languages such as PHP (e.g., BugBox [26]), Java (e.g., WebGoat [27]) and Server-Side JavaScript (SSJS) (e.g., TestREx [28]). This still tremendously helps in understanding insecure coding practices and (anti-)patterns in those languages. However, such corpora do not exist for Lua. This limits the general understanding of how to securely code in Lua, or how to detect and prevent insecure Lua coding practices in (preferably) automated ways.

In summary, our main contributions with this work are:

- We develop and open-source the first static analysis tool for Lua code that mainly focuses on finding most common security issues.
- We build and open-source the first public corpus of synthetic Lua code samples containing both vulnerable and non-vulnerable examples. For example, it can be used as a baseline for tool evaluation/comparison or for secure coding practices.
- We release the testing setups used in our experiments in form of virtual and reproducible environments.

The rest of this paper is organized as follows. In Section II we start with summaries of the most notorious uses of Lua, which in our opinion are useful for the overall picture and to further motivate our work. Section III overviews Lua technologies relevant to our work and Section IV describes our experimental setup. We detail the covered vulnerabilities and the vulnerable Lua code corpus in Section V. Then, in Section VI we describe the design, implementation and preliminary results. We discuss related work in Section VII, and conclude with Section VIII.

II. Notorious Lua Usage

Several web directories aggregate regularly updated lists of projects, products and companies that use Lua one way or another [29], [30], [14], [31]. In addition to these web directories, the following Google search query can help discover other (sometimes surprising) uses of Lua:

```
Copyright Lua PUC-Rio filetype:PDF
```

We strongly encourage the interested readers to explore and contribute to these web directories. However, we will focus on summarizing the most notorious (at least from our perspective) Lua use cases. We believe that providing such a summary strongly adds to the motivation behind our work.

A. Web and Internet Projects

Wikipedia added Lua as a templating language [8]. CloudFlare is using Lua code with Nginx server to power many of the services and modules that run within their infrastructure [9]. GitHub re-engineered their GitHub Pages service to efficiently run with load-balanced Nginx instances that run a simple yet effective Lua module to route requests to hosting filesystems [10]. TaoBao, the largest e-commerce site in Asia and in Top 100 in Alexa ranking, serves its content using `Tengine` [32]. TaoBao develops `Tengine` which is an Nginx-forked web-server that also supports Lua dynamic scripting language. Lua is also used at incredible online scale for example in Massive Multiplayer Online Role-Playing Games (MMORPG) such as World of Warcraft (WoW) [33].

B. Projects, Tools and Utilities

Nmap provides Nmap Scripting Engine (NSE) [34], which is one of the Nmap's most powerful and flexible features. It allows its users to write scripts to automate a wide variety of networking tasks and for this it embeds a `Lua5.2` interpreter. Wireshark has an embedded Lua interpreter [35] and allows writing dissectors, post-dissectors and taps in Lua. Both Wireshark and Nmap provide Lua interpreter to allow quick development of network analysis tools. However, if not written with security in mind those Lua scripts can be attacked by malicious network nodes and traffic using for example log poisoning, command and code injection. Google StreetView team, and Russ Smith primarily, developed *lua-checker* to validate their Lua scripts [19]. Syhunt is using Lua as part of their security tools for web applications. They also recently started using Lua modules for web applications within their Apache and Nginx web servers [36]

C. Security Incidents and Malware

Below we summarize publicly known instances where malware, both classical and embedded/IoT, used Lua one way or another. Since these malware instances (partly) contain Lua code, it is perhaps possible to find vulnerabilities within the malware itself using our tool. Finding vulnerabilities within malicious code can potentially help the cyber-protection organizations in their efforts of limiting the impact of such attacks, and taking down the malware and the botnets [37].

1) Conventional Malware:

a) Flamer: (a.k.a. Flame, a.k.a. sKyWIper, a.k.a. Skywiper) [38] is one of the most sophisticated pieces of malware analyzed to that date. It was compared to *Stuxnet* and *Duqu* in terms of complexity and its use in mainly targeted attacks. *Flamer* is described as an info-stealer malware with highly modular structure that incorporates multiple propagation and attack techniques. One important piece of this malware is the `mssecmgr.ocx` file which contained a Lua interpreter along with SSH code, and SQL functionality. The Lua interpreter makes this component highly flexible and configurable, allowing the attackers to deploy updated commands and functionality quickly and efficiently. Crysys Lab released a comprehensive list of all Lua modules (pre-compiled `.luac` files) present within the malware, and detailed some interesting ones such as `CRUISE_CRED.lua` which collects credential information from an already infected machine [39]. The malware name comes from the `FLAME` prefix in some of the variable names in Lua modules, such as `FLAME_ID_CONFIG_KEY`.

b) EvilBunny: was analyzed in 2014 by Marion Marschalek at Cyphort [40]. It was described as a technically fascinating piece of malware, similar to the targeted attacks samples seen in the wild around 2011, such as *Stuxnet*, *Duqu* and *Flamer*. *EvilBunny* is a sophisticated malware that also aims to trick sandboxes and performed quite uncommon tricks

to evade detection. In addition, it was specifically designed to be an execution platform for Lua scripts that can be injected by the attacker at any stage of the attack. It embeds a `Lua5.1` interpreter, which allows it to download and execute Lua scripts in order to reach a certain level of polymorphism. The design of *EvilBunny* is such that Lua scripts can call back into the malware's C++ code to modify its behavior during runtime.

c) ProjectSauron: (a.k.a. *Strider*) was analyzed by Kaspersky [41] based on their detection in 2015 of the massive activity from a new threat actor they codenamed *ProjectSauron*. The threat actor is supposedly responsible for large-scale attacks against key governmental organizations in several countries. Symantec estimates this malware was active since 2011 [42]. *ProjectSauron* embeds a Lua interpreter and includes main Lua modules such as Network Loader, Host Loader and Keylogger. The malware name comes from the `SAURON` prefix in some of the variable names in Lua modules, such as `SAURON_KBLOG_KEY`.

2) Malware for IoT and Embedded Devices:

a) LuaBot: is a malware for embedded/IoT devices, and was initially discovered and analyzed by MalwareMustDie researchers who dissected its ARMEL version [43]. At the same time, the researchers at Symantec analyzed its ARMEB variant [44]. MalwareMustDie researchers suggest that it contains a `Lua5.3` interpreter along with around two dozens of Lua scripts. This malware abuses some vulnerabilities in cable modems disclosed by Bernardo Rodrigues [45], who also confirmed that the malware author packed the Lua scripts of the malware as a GZip blob. This malware puts the infected devices into botnets which are used in large-scale attacks, similar to the Mirai DDoS attack [46] which abused multiple vulnerabilities in web-cameras, CCTV and video surveillance systems [47].

D. IoT and Embedded Devices

Lua is a core component of several mainstream firmware and IoT/IIoT platforms. OpenWrt [48] can be safely called a "de facto" standard distribution for IoT and embedded platforms. It is powering classical networking devices (e.g., routers and broadband modems [49]), IoT-focused boards (e.g., *LinkIt Smart 7688* [50]), and other IoT-focused distributions (e.g., Linino [51]). OpenWrt heavily uses Lua Configuration Interface (LuCI) [52]. LuCI uses the Lua programming language with its object-oriented libraries and template features to provide a clean, easy to extend and maintainable web user interface for embedded and IoT devices [53]. Lua is also a core API component inside Wind River's (now Intel's) Intelligent Device Platform (IDP), which includes a Lua VM as depicted in the components description figure of IDP product note [54]. Finally, there is an exploding market and an overwhelming popularity of extremely low-cost IoT chipsets such as ESP8266 [11] that can run Lua code using open-source firmware such as NodeMCU [12] and NodeLua [13].

III. Overview of Relevant Lua Ecosystem

Below we provide an overview of Lua technologies that are relevant to the scope of our work and experiments.

A. Lua Interpreters

Besides its syntax and semantic specifications Lua language is supported by Lua interpreters, and several popular Lua interpreters exist. There is the original Lua implementation [29] developed at PUC-Rio which releases version specific interpreters such as `Lua5.1, Lua5.2, Lua5.3`. These are usually readily available as packages in major Linux distributions. `LuaJIT` [55] is an alternative and full implementation of Lua interpreter using a Just In-Time (JIT) compiler. It was developed for applications that require additional speed and performance gains. `eLua` [56] stands for Embedded Lua, and is an alternative and full implementation of Lua language, tailored to the world of embedded devices. It extends Lua with specific features for efficient and portable embedded software development. `LLVM-Lua` [57] is a JIT static Lua compiler. It uses Low Level Virtual Machine (LLVM) as its JIT compiler backend. In LLVM-Lua case, the Lua scripts are compiled to native code using LLVM's JIT engine and support natively any CPU architecture supported by LLVM executables.

B. Web Technologies and Lua

In order to run Lua code in the context of web servers, web applications and web services, there must exist support and bindings for Lua that can integrate and interpret its code in the web context. The `mod_lua` package is an external module for the popular `Apache2` httpd server. This module provides Lua hooks into various portions of the `Apache2` httpd request processing. It basically allows the `Apache2` web server to be extended with scripts written in Lua [58]. `OpenResty` provides a full-fledged web platform. It integrates the standard `Nginx` core, the Lua interpreter (e.g., `LuaJIT` or standard Lua) through the `ngx_lua` module, and a wide range of well-written Lua libraries. It also allows Lua web applications to access the Nginx core, 3rd-party Nginx modules, and most of their external dependencies [59]. At very high level, it is a very convenient packaging of Nginx [60] that contains the ngx_lua [61] module with bindings and configurations for standard Lua or `LuaJIT`. The `ngx_lua` package is the Nginx module that integrates Lua code interpretation into the Nginx core. Using standard Lua interpreter or `LuaJIT`, it embeds Lua into Nginx. It leverages Nginx's subrequests, and thus allows the integration of the powerful and efficient Lua threads (i.e., Lua coroutines) into the Nginx event model [62]. The `CGILua` [63] package implements the CGI/FCGI protocols. The Common Gateway Interface (CGI) and Fast-CGI (FCGI) are standard protocols for web servers to execute binary and scripting console programs that generate and process web pages dynamically. `CGILua` allows creating web pages dynamically and processing of web forms data. It also separates the logic and data handling for web page generation, thus simplifying the development of web applications in Lua. `CGILua` provides an abstraction of the underlying web server.

Therefore, the advantage of CGILua is that by following the CGI specifications, it is easy to integrate it into any compatible web server that implements the CGI/FCGI interfaces, such as Apache2, Nginx/OpenResty, Lighttpd. The Lua-WSAPI [64] package implements the Web Services API (WSAPI). The WSAPI is an API that abstracts the web server from the web applications. The advantage of Lua-WSAPI code is that by following the WSAPI interface, it is easy to integrate such code it any WSAPI-compatible web server that implements the CGI/FCGI interfaces, such as Apache2, Nginx/OpenResty, Lighttpd. Currently Lua-WSAPI provides the wsapi-cgi and wsapi-fcgi packages that run over the CGI and FCGI, respectively.

It must be pointed out that Scott and Sharp [65] indicate that a web application regardless of its underlying technology will always be vulnerable, and therefore Lua is no exception to this assumption.

IV. Our Experimental Setup

We performed our setup on a virtual and reproducible environment (e.g., using Vagrant [66]) having the following OS configuration as in Listing 1:

```
lua@lua-re:~$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04.5 LTS"

lua@lua-re:~$ uname -a
Linux lua-re 4.2.0-27-generic #32~14.04.1-Ubuntu SMP
Fri Jan 22 15:32:27 UTC 2016 i686 i686 i686 GNU/Linux
```

Listing 1. Base configuration of our Ubuntu testing environment.

We configured this environment to host both Lua5.1 and Lua5.2 interpreters. We also configured and installed a version of luarocks package manager for each interpreter version, i.e., luarocks-5.1 and luarocks-5.2 respectively [4]. In our testing and evaluation we used only Lua5.2, but the concept is similarly applicable to the Lua5.1 configuration which is readily available within the environment.

We have chosen the following four configurations to host Lua-based web applications. We believe they provide a sufficiently good coverage of existing technologies and setups. These configurations of course can be easily expanded in the future work to provide even more setup variety.

A. Apache2 with mod_lua code

First, we installed the default and basic Apache2 packages. Then, we used the Apache2 development packages to compile and install the mod_lua with bindings to Lua5.2. To configure mod_lua support in Apache2, we added the following snippet to the /etc/apache2/sites-enabled/000-default.conf as in Listing 2:

```
AddHandler      lua-script .mod_lua
LuaScope        thread
LuaCodeCache    stat
AcceptPathInfo  On
```

Listing 2. Enabling mod_lua support in Apache2.

[4]Similar to pip2 for Python2 and pip3 for Python3, respectively.

B. Apache2 with CGILua code

For Lua samples based on the CGILua package, we installed the CGILua package using the Lua5.2's corresponding luarocks-5.2. We reused the same Apache2 instance as above and configured CGILua code to be run in FastCGI (FCGI) mode. To configure CGILua support in Apache2, we added the following snippet to the /etc/apache2/sites-enabled/000-default.conf as in Listing 3:

```
<Directory "/var/www/html/CGILua">
AllowOverride All
Options ExecCGI
AddHandler fcgid-script .lua
AddHandler fcgid-script .lp
FCGIWrapper /usr/local/bin/cgilua.fcgi .lua
FCGIWrapper /usr/local/bin/cgilua.fcgi .lp
</Directory>
```

Listing 3. Enabling CGILua support in Apache2.

C. Nginx with ngx_lua code

For Lua code using ngx_lua API, we installed the *OpenResty* package and its dependencies. This allows us to create Nginx configuration files that contain Lua code for the web service. The simplest Lua code snippet that uses Nginx and ngx_lua is presented in Listing 4:

```
worker_processes  1;
error_log logs/error.log;
events {
    worker_connections 1024;
}
http {
    server {
        listen 8080;
        location / {
            default_type text/html;
            content_by_lua '
                ngx.say("<p>hello, world</p>")
            ';
        }
    }
}
```

Listing 4. Enabling Lua code in Nginx and ngx_lua.

Then we use these configuration files to actually start the Nginx web server that will run the Lua code from the configuration files. This can be accomplished with a simple command similar to the one in Listing 5:

```
/usr/local/openresty/nginx/sbin/nginx -p `pwd`/ -c nginx.conf
```

Listing 5. Starting Nginx web server with configuration file for Lua code support.

D. Lighttpd with vanilla Lua code

To run vanilla Lua code within a web server, we installed WSAPI Lua package via `luarocks-5.2`. We also installed the standard `Lighttpd` package with its dependencies, and enabled CGI and FCGI modules in its configuration. To configure Lua WSAPI CGI support in `Lighttpd`, we added the following snippet to the `/etc/lighttpd/conf-enabled/10-cgi.conf` as in Listing 6:

```
cgi.assign      = (
        ".lua"  => "/usr/local/bin/wsapi.cgi",
)
```

Listing 6. Enabling vanilla Lua code to server web requests in Lighttpd via WSAPI CGI.

E. Smoke Testing

All these environments undergo a "smoke test" in order to insure they are up-and-running, and they work as expected with Lua bindings. Also, we test that the vulnerable samples are indeed exploitable and the non-vulnerable do not exploit the system.

V. Vulnerabilities and Samples

Daragon [36] first summarized a set of Lua code examples that intentionally contain security vulnerabilities, mainly from *OWASP Top10* . Their report is comprehensive and provides great details about potential security pitfalls in Lua code. However, it falls short of full practical usability for several reasons. First, the vulnerable Lua code samples are not compiled into a ready-to-run and ready-to-test corpus. Second, the report leaves to the user the burden of actually creating *a reproducible environment* (i.e., the working configurations and the code samples) in order to actually test the vulnerabilities and the proposed countermeasures. This in turn can be counterproductive to perform in practice, and can be prone to errors as well. Therefore, our work in a way is a natural and practically-applicable extension to their work.

Below we present a sample of Lua code for each vulnerability class that is supported by our tool. Since we support multiple Lua configurations, as detailed in Section IV, we will present here code for a single configuration. Nevertheless, the corpus we open-source contains the Lua code samples for all configurations.

A. Cross-Site Scripting (XSS)

Listing 7 presents Lua code for `CGILua` vulnerable to reflected XSS attacks. The original intent of this Lua code is to welcome a logged-in user with her username in the page header.

```
#!/usr/bin/env wsapi.fcgi

local luatech = 'CGILua'
local username = cgilua.QUERY.username or 'Unknown username'

cgilua.htmlheader()
cgilua.put([[
<title>XSS Lua ]] .. luatech .. [[</title>
]])
cgilua.put([[ Hello, ]] .. username)
```

Listing 7. CGILua code containing a simple XSS vulnerability.

This vulnerability can be triggered for example by accessing the following URL: http://lua-VM/cgilua/xss/xss_reflected_true.lua?username=%3Cscript%3Ealert(1);%3C/script%3E. In turn, this will perform a reflected XSS inside the browser of a victim user who is tricked into opening the malicious URL.

B. OS Command Injection

Listing 8 presents Lua code for `CGILua` vulnerable to OS command injection attacks. The original intent of this Lua code is to directory list the home folder of a particular user.

```
#!/usr/bin/env wsapi.fcgi

local luatech = 'CGIlua'
local username = cgilua.QUERY.username or 'root'

cgilua.htmlheader()

cgilua.put([[
<title>OS Command Injection Lua ]] .. luatech .. [[</title>
]])

local handle = io.popen("ls -lart /home/" .. username)
local data = handle:read("*a")
handle:close()

cgilua.put([[
Listing home dir:
]] .. data)
```

Listing 8. CGILua code containing a simple OS command injection vulnerability.

This vulnerability can be triggered for example by accessing the following URL: http://lua-VM/cgilua/os-command-injection/os-command-injection_os.execute_true.lua?username=lua;ls-lart/etc. In turn, this will also list the contents of the `/etc` directory.

C. Null Byte Injection

Listing 9 presents Lua code for `CGILua` vulnerable to null byte injection attacks. The original intent of this Lua code is to list to the user browser contents of *only* text files, therefore the assumption that appending `".txt"` will work. However, null byte injection is successful in this case.

```
#!/usr/bin/env wsapi.fcgi

local luatech = 'CGIlua'
local textfile = cgilua.QUERY.textfile or 'README'

local f = io.open(textfile .. ".txt")
local data = f:read("*a")
f:close()

cgilua.htmlheader()

cgilua.put([[
<title>Null-Byte Injection Lua ]] .. luatech .. [[</title>
]])

cgilua.put([[
Your text file contains:
]] .. data)
```

Listing 9. CGILua code containing a simple null byte injection vulnerability.

This vulnerability can be triggered for example by accessing the following URL: http://lua-VM/cgilua/null-byte-injection/null-byte-injection_true.lua?textfile=/etc/passwd%00. In turn, this will list the contents of `/etc/passwd` file (or eventually other sensitive file).

D. Path Traversal

Listing 10 presents Lua code for `CGILua` vulnerable to path traversal attacks. The original intent of this Lua code is to list to the user browser the contents of *only* files within `/tmp` folder, therefore the assumption that prepending `"/tmp/"` will work. However, path traversal is successful in this case.

```
#!/usr/bin/env wsapi.fcgi

local luatech = 'CGIlua'
local tmpfile = cgilua.QUERY.tmpfile or '.X0-lock'

local f = io.open("/tmp/" .. tmpfile)
local data = f:read("*a")
f:close()

cgilua.htmlheader()

cgilua.put([[
<title>Path Traversal Lua ]] .. luatech .. [[</title>
]])

cgilua.put([[
Your tmp file contains:
]] .. data)
```

Listing 10. CGILua code containing a simple path traversal vulnerability.

This vulnerability can be triggered for example by accessing the following URL: http://lua-VM/cgilua/path-traversal/path-traversal_true.lua?tmpfile=../etc/passwd. In turn, this will list the contents of `/etc/passwd` file (or eventually other sensitive file).

E. Local File Inclusion

Listing 11 presents Lua code for `CGILua` vulnerable to path traversal attacks. The original intent of the code is to load a particular Lua Pages template, for example when the user selected a specific foreign language in the web application.

```
#!/usr/bin/env wsapi.fcgi

luatech = 'CGIlua'
local template = cgilua.QUERY.template or "default_template"

cgilua.htmlheader()
cgilua.handlelp(template .. ".lp")
```

Listing 11. CGILua code containing a simple local file inclusion vulnerability.

This vulnerability can be triggered for example by accessing the following URL: http://lua-VM/cgilua/lfi/lfi_handlelp_true.lua?template=/var/www/html/admin/cgilua/reboot.lp%00
In turn, this will include for execution a file (in this case a CGILua Lua Pages) that could potentially execute high-privileged commands. The `/var/www/html/admin/cgilua/reboot.lp` file could simply contain a call to OS such as `<?os.execute("reboot")?>`.

F. SQL Injection

Listing 12 presents Lua code for `CGILua` vulnerable to SQL injection attacks. The original intent of this code is to check the credential of a user trying to log into the web application, and to display a message in case both the username and password match.

```
#!/usr/bin/env wsapi.fcgi

local luatech = 'CGIlua'
local u = cgilua.QUERY.username or ''
local p = cgilua.QUERY.password or ''
local logged = 0

local mysql = require "luasql.mysql"
local env  = mysql.mysql()
local conn = env:connect('mysqldb', 'mysqluser', 'mysqlpass')

cur, err = conn:execute("select * from users where
    username = '" .. u .. "' and password = '" .. p .. "'")

row = cur:fetch ({}, "a")

while row do
    row = cursor:fetch (row, "a")
    logged = 1
end

cgilua.htmlheader()

cgilua.put([[
<title>SQL Injection Lua ]] .. luatech .. [[</title>
]])

if logged == 1
then
    print("Yes")
    cgilua.put([[You successfully logged in as ]] .. u)
end

cursor:close()
conn:close()
env:close()
```

Listing 12. CGILua code containing a simple SQL injection vulnerability.

This vulnerability can be triggered for example by accessing the following URL: http://lua-VM/cgilua/sqli/sqli_mysql_true.lua?username=super_admin_god';--&password=does_not_matter

VI. Our Tool

A. Summary of Design and Implementation

Our tool has three logical parts and we detail them below.

1) Parser: The first part is the parser which parses Lua code and creates the parse tree, also known as the Concrete Syntax Tree (CST). To create the parser in our tool we used ANTLR4 [67] which is a tool that generates parsers based on LL(*) (LL-regular) parsers. ANTLR4 requires a language grammar specification in a particular format, namely `.g4`. Conveniently enough, the Lua project provides the specification in extended Backus-Naur Form (BNF) [5]. And in fact, there is already a reference specification in `.g4` format for Lua, based on its BNF specification [6]. We use and adapt this specification to be able to parse Lua code used in our synthetic samples as well as in real-world projects as found

[5]See https://www.lua.org/manual/5.3/manual.html

[6]See https://github.com/antlr/grammars-v4/blob/master/lua/Lua.g4

for example on GitHub [68], [69]. Alternatively, one could use the parsers from the existing tools, such as LuaCheck [17] or lua-checker [19]. Unfortunately the parsers in those tools are tailored to particular versions of Lua specifications and adapting them to all or latest Lua specifications may be untrivial or may introduce bugs and unnecessary complexity. Therefore, we argue that specifying the Lua syntax and grammar via a clean BNF specification written in a `.g4` file format is a future-proof approach for a maintainable project.

```
$ antlr4 -Dlanguage=Python2 Lua.g4
$ ls
LuaLexer.py
LuaParser.py
LuaListener.py
LuaVisitor.py
```

Listing 13. Compilation of the parsing modules of our tool using Python2 as the target for the parser implementation.

Subsequently, we extend the `LuaListener` stub class with the necessary functionality (e.g., `class LuaListenerExtended(LuaListener):`) to walk the parse tree in order to create the Abstract Syntax Tree (AST) based on it.

2) Abstract Syntax Tree Generator: The second part is the Abstract Syntax Tree (AST) generator, which transforms the CST generated by the parser into a tree of basic blocks that provide enough computational abstraction to apply security and taint analysis. We then create the AST by walking the parse tree which the ANTLR4-generated parser will create in-memory. Listing 14 shows the simplified code that will start the parse-tree walking and AST generation:

```
(...)
ASTgenerator = LuaListenerExtended()
walker = ParseTreeWalker()
walker.walk(ASTgenerator, tree)
(...)
```

Listing 14. Walking the parse tree of of analyzed Lua code and generating its AST representation.

3) Security Analyzer: The third part is the security analyzer (scanner) itself. It walks the AST and performs the taint propagation and analysis on the AST. This way it is able to detect potential vulnerable blocks that allow dangerous tainted inputs from the user into the sensitive sinks. Additionally, we implement simplest context parsers that can potentially help the security analyzer to understand whether the code context opens up for a vulnerability. One example is HTML markup parser for sinks such as `print` and `cgilua.put` that can try to decide whether a tainted input requires a single-quote or a double-quote in order to successfully inject an XSS. Another example is the OS commands context parsers for sinks such as `os.execute` and `io.popen` that can try to decide whether a tainted input requires a single-quote or a semicolon in order to achieve a OS command injection/execution. Similarly to the above we have a SQL context parser, with similar goals and simple heuristics as above. The security analyzer makes extensive use of the list of tainted input sources and sensitive sinks. We have systematically summarized the tainted input sources and sensitive sinks (both exposed by vanilla Lua specifications as well as exposed by other packages/modules) that are relevant to Lua code and that our tool supports. Due to space considerations we skip this list from the submission, however it can be thoroughly consulted in the configuration files of the tool itself.

B. Preliminary Results

Our tool is at a proof-of-concept stage. It has still to undergo heavy testing and evaluation using more complex samples and real-world projects, Also it has some limitations which we discuss in Section VI-D.

However, at this stage it is able to successfully detect 100% of all the simple true positive vulnerable samples that we detailed in Listings 7, 8, 9, 10, 11, 12, regardless of the configuration under test as described in Section IV.

C. Tool's Additional Uses

As already mentioned, Lua's popularity is on the rise in the last couple of years. There is already a considerable amount of open-source Lua code [68], [69], and we expect it to grow [7]. The amount and the open-source nature of the such Lua code prompts consideration for potential software license violations. There are many techniques to find such code-clones [70] and software license violations [71]. The Binary Analysis Toolkit (BAT) [72], previously known as GPLtool, is one such framework that allows detection of potential software license violations, besides many other things. For this to be possible, BAT needs a database of distinguishable source-code attributes that identify a particular source-code sample. For example, such attributes could be file names, function names, variables names, string constants. We implemented in our tool a special mode of operation that outputs the main elements of Lua code (e.g., function names, variables names, string constants) as structured output such as JSON and XML. This mode of operation takes advantage of the presence of the Lua parser and the availability of the parse-trees. This allows tools such as BAT to easily support Lua code and import Lua source-code information into their database for further processing (e.g., detection of compliance, violation or cloning).

D. Limitations and Future Work

Unfortunately, our tool currently has some limitations that we intend to address as soon as possible in our future work.

1) Second-order Vulnerabilities: For example, our tool cannot yet detect more complex variants of the discussed vulnerability classes, such as *second-order vulnerabilities*. Second-order vulnerabilities occur when the exploitation payload is first stored by the vulnerable application (e.g., web server or web application), and then later on uses it a security-sensitive operation. Our tool lacks the detection of second-order vulnerabilities as this is not trivial. One possible way to overcome this limitation is to use the static analysis approach proposed by Dahse and Holz [73] and implemented into RIPS

[7] One example is the exploding market of low-cost IoT devices (such as ESP8266) running open-source Lua firmware (such as NodeMCU [12] and NodeLua [13]).

static analysis tool for PHP [74]. Another possibility is to use the Lua ASTs that our tool generates in conjunction with (extended) Joern [75], which is a generic framework for source code vulnerability discovery using AST [76], [77], [78].

2) Limitations of Static Analysis: Our tool requires also a configuration input specifying which input-sanitization functions should be treated as safe and which should be not. This is needed because it is non-trivial to determine whether such functions are indeed safe using pure static approaches as highlighted for example by Balzarotti et al. [79].

For example, Listing 15 implements a safe HTML input-sanitization function, described by Daragon [36]. It replaces "dangerous characters" in an input with their safe HTML equivalents.

```
function htmlescape(text)
local special = {
['<']='&lt;', ['>']='&gt;', ['&']='&amp;', ['"']='&quot;'
}
return text:gsub('[<>&"]', special)
end
```

Listing 15. Safe input-sanitization function.

However, the input-sanitization function in Listing 16 is unsafe. Though it very much resembles the safe input-sanitization function in Listing 15, the subtle difference is that the "dangerous characters" to be matched and escaped are preceded with spaces. As a consequence, the `text:gsub` will not replace the actual "dangerous characters" with their safe HTML equivalents, such as `&lt` and `&amp`. Such simple yet easy to overlook vulnerabilities could in theory be introduced sometimes by unintended mistakes [8], and other times by potentially malicious intent [9].

```
function htmlescape(text)
local special = {
[' <']='&lt;', [' >']='&gt;', [' &']='&amp;', [' "']='&quot;'
}
return text:gsub('[<>&"]', special)
end
```

Listing 16. Unsafe input-sanitization function.

Another example relates to the analysis of the vulnerable context. For instance, an XSS attack makes sense in an JS/HTML context (e.g., browser parsing and rendering), but will not be applicable in a plain-text context. A static analysis tool can in principle perform only very basic context parsing and analysis. For example, when detecting that a tainted input source from the `GET` request *directly reaches* a sensitive sink such as a printout function (e.g., `ngx.say` or `r:puts(r:parseargs().username)`), it could analyze the `content-type` (e.g., `ngx.header.content_type = "text/html"` or `r.content_type = "text/plain"`). If there is a match for `"text/html"` it marks the XSS as a true positive. However, if there is a match for `"text/plain"` it marks the XSS as a non-issue. Finally, if `r.content_type = base64.decode("dGV4dC9odG1s")` [10] such static analysis and matching will simply not work or becomes absolutely non-trivial.

One way to overcome the type of limitations described above is to use a particular application of the *abstract interpretation* technique [80], called *abstract parsing*. Abstract parsing [81] is a program pre-execution static technique to analyze the documents (or outputs in general) generated dynamically by a program block. It was demonstrated to *statically validate* a suite of PHP programs (i.e., dynamic interpreted language programs) that dynamically generate HTML files. Another way to overcome such limitations of the input-sanitization uncertainty is by using a hybrid approach. That is, drive and focus the dynamic analysis onto this particular part of code that was initially marked as suspicious or unsure by the static analysis, as proposed by Balzarotti et al. [79] or as implemented in [25]. We leave the evaluation and the implementation of such enhancements for future work.

3) Support for Lua Pages Code: Additionally, our tool does not currently support code written in Lua Pages. Lua Pages are supported only by the `CGILua` setups. In simple terms, they are similar to PHP or Java Server Pages (JSP), but intended for Lua server-side engines. They represent (X)HTML pages with Lua code embedded in-between using escape sequences such as `<?lua` and `?>`. For example, Listing 17 shows the Lua Pages equivalent of the pure Lua code vulnerable to XSS presented in Listing 7.

```
<?lua
local luatech = 'CGILua LuaPages'
local username = cgilua.QUERY.username or 'Unknown username'
?>
<title>XSS Lua <%= luatech %> </title>
Hello World, <%= username %>
```

Listing 17. Lua Pages code containing an XSS vulnerability.

We estimate it should be fairly straightforward to write a "Lua Pages"-to-"Lua code" transform by employing some (X)HTML parsing [11] and parse-tree processing using ANTLR4 [67] for example. For example, such a transform could process the Lua Pages code from Listing 17 and produce an equivalent pure Lua code shown in Listing 18. Following this transform, our tool would be used on the transformed equivalent code similarly to other Lua code.

```
cgilua.htmlheader()

local luatech = 'CGILua LuaPages-to-Lua'
local username = cgilua.QUERY.username or 'Unknown username'

cgilua.put([[<title>XSS Lua ]])
cgilua.put(luatech)
cgilua.put([[ </title>
Hello World, ]])
cgilua.put(username)
```

Listing 18. Equivalent Lua code produced via transform from the previous Lua Pages example that contained an XSS vulnerability.

[8] https://github.com/MrMEEE/bumblebee-Old-and-abbandoned/issues/123

[9] See statements that Juniper discovered unauthorized code in ScreenOS https://kb.juniper.net/InfoCenter/index?page=content&id=JSA10713&actp=search

[10] `dGV4dC9odG1s` string is the base64 coding of `text/html` string.

[11] https://github.com/antlr/grammars-v4/blob/master/html/HTMLParser.g4

Finally, we plan to validate and improve our tool more thoroughly by running it on large and mature projects developed in Lua.

VII. Related Work

A. Lua Security

Daragon [36] first summarized and described the risks for Lua application as posed by most common vulnerabilities, such as the ones from *OWASP Top10* . They also suggested some simple protective countermeasures, and discussed configuration and deployment hardening. We take their work one step further and, in addition to developing and open-sourcing our Lua static analysis tool, we build and open-source the samples and the setup into an easy to reproduce, extend and share environment.

B. Static Analysis of Dynamic and Interpreted Languages

There is a large amount of work done in the area of static analysis and in particular static analysis of dynamic, interpreted and scripting (web-oriented) languages. Huang et al. [82] presented a sound approach to ensure security of web applications. The authors created a lattice-based static analysis algorithm derived from type systems and typestate. They also implemented their approach into the `WebSSARI` static analysis tool, which however supports PHP programs. Di Lucca et al. [83] proposed a static analysis approach for testing web applications in order to detect XSS vulnerabilities. Then they propose to cross-check the analysis results with dynamic testing techniques in order to eliminate false positives. They have demonstrated the effectiveness of their approach on real-world web applications implemented in PHP and ASP. Jovanovic et al. [84] proposed a flow-sensitive, interprocedural and context-sensitive data flow static analysis to find vulnerable locations in web applications. The `Pixy` static analysis tool is the prototype implementation of their approach that is targeted to discover XSS vulnerabilities in PHP scripts. Xie and Aiken [85] presented a static analysis algorithm that discovers security vulnerabilities in PHP projects. Their analysis algorithm employed a novel three-tier architecture to capture information at the intra-block, intra-procedural, and inter-procedural levels. Wassermann and Su [86] presented a static analysis technique to discover XSS vulnerabilities. Their approach directly addressed weak or absent input validations, and combined ideas from tainted information flow with string analysis. The authors evaluated their approach on several real-world PHP web applications. Livshits and Lam [87] proposed a static analysis approach to detect common web application vulnerabilities such as SQL injections, XSS, and HTTP splitting. Their approach is based on a scalable and precise "points-to analysis", and was shown to detect a number of important vulnerabilities in real-world Java projects. Biggar and Gregg [88] presented an improved static analyzer for PHP applications, by extending classical static analysis to analyse more complex PHP code. Their analysis approach combined alias analysis, type-inference and constant propagation for PHP. Dahse and Holz [73] proposed a static code analysis approach to discovering second-order vulnerabilities in web applications. In their approach the authors analyzed reads and writes to memory locations and identified unsanitized data flows by connecting input/output data blocks of persistent data stores. Additionally, Dahse and Holz [89] proposed a static analysis based vulnerability discovery approach in PHP programs by precisely modeling the highly dynamic nature of PHP code. The authors configured and simulated over 900 PHP built-in features to create the model, and then performed inter- and intra-procedural data flow analysis by creating block and function summaries. They implemented both their static analysis approaches into a prototype called RIPS [74] which supports only PHP web applications.

However, none of these works so far have taken Lua as their target or have addressed specific Lua security issues. In this regard, our work is complementary to the described works and fills the void within this research space.

C. Corpora of Vulnerable Code Samples

Contrary to the field of static analysis techniques and tools where a lot of work has been done, the area of vulnerable code corpora that is systematically collected/synthesized and well-maintained is less generous. We are aware of only several such projects as follows. OWASP develops and maintains WebGoat [27], [90], a project that contains a collection of synthesized Java and J2EE web modules that are deliberately insecure. It was designed to teach web application security and practically demonstrate common server-side application flaws. WebGoat also provides virtualized reproducible environments such as Vagrant and Docker. Nilson et al. [26] introduced BugBox, a corpus of real-world PHP vulnerable web applications and an exploit simulation environment. BugBox [91], [92] provides virtualized reproducible environments and packaging mechanisms to share and distribute vulnerability and exploit data. The goal of BugBox is to facilitate the empirical studies of vulnerabilities, evaluation of security tools, and research on security metrics. Dashevskyi et al. [28] argued that in order to test if existing exploits can be reproduced in different settings and to facilitate discovery of new vulnerabilities, reliable testbeds are required. Therefore the authors presented TestREx [93] testbed for repeatable exploits. TestREx allows packaging and running applications along the intended environment, injecting and monitoring exploits, and generating security reports. Within TestREx, the authors also packaged a corpus of vulnerable SSJS node.js samples they developed, along with existing corpora such as WebGoat [27], [90] and BugBox [91], [92].

By building and open-sourcing the corpus of both vulnerable and non-vulnerable Lua code samples, our work supplements the relatively scarce research corpora in this field that are available to security researchers, practitioners and evaluators. Additionally, our work aims to facilitate empirical vulnerability studies, practical teaching of (in)secure coding patterns in Lua, and evaluation of static and dynamic security tools for Lua and web applications.

VIII. Conclusion

Lua is a powerful and performant dynamic language. Its popularity is on the rise within the embedded/IoT applications, as well as within large organizations for their web platforms. However, there is a lack of both static analysis tools for Lua code and corpora of vulnerable Lua code samples.

In this paper we present the first public Static Analysis for Security Testing (SAST) tool for Lua code that is currently focused on web vulnerabilities. We also present and release our synthesized corpus of intentionally vulnerable Lua code, as well as the virtualized testing setups used in our experiments. At this preliminary stage our tool detects all 100% of the true positives that we detailed in Listings 7, 8, 9, 10, 11, 12 We hope our work can spark additional and renewed interest for source code static analysis and Lua security.

Acknowledgements

The authors thank the NLnet Foundation (http://www.nlnet.nl) and the Binary Analysis Tool (BAT) project (http://www.binaryanalysis.org) for their financial support under the NLnet project number *2014-09-017e*. We extend our thanks personally to Armijn Hemel from Tjaldur Software Governance Solutions (http://www.tjaldur.nl) and Michiel Leenaars from NLnet foundation (http://www.nlnet.nl) for having confidence in this project's usefulness and potential, and for their continuous support and trust. Last but not least, we thank all the anonymous reviewers for their valuable time and inputs, and for the constructive feedback that helped us improve our paper and results.

References

[1] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, "Lua – an extensible extension language," *Journal of Software: Practice and Experience*, vol. 26, no. 6, pp. 635–652, 1996. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-024X(199606)26:6⟨635::AID-SPE26⟩3.0.CO;2-P

[2] "TIOBE Index," Jan 2017, (Accessed: Jan 2017). [Online]. Available: http://www.tiobe.com/tiobe-index/

[3] "PYPL – PopularitY of Programming Languages Index," Jan 2017, (Accessed: Jan 2017). [Online]. Available: http://pypl.github.io/PYPL.html

[4] P. Chapuis, "State of the Lua Ecosystem," Nov 2013, (Accessed: Jan 2017). [Online]. Available: https://www.lua.org/wshop13/Chapuis.pdf

[5] "CodeEval – Most Popular Coding Languages of 2015," Feb 2015, (Accessed: Jan 2017). [Online]. Available: http://blog.codeeval.com/codeevalblog/2015

[6] "Performance comparison of LuaJIT against other VMs on different architectures," (Accessed: Jan 2017). [Online]. Available: http://luajit.org/performance.html

[7] "The Computer Language Benchmarks Game – Lua," (Accessed: Jan 2017). [Online]. Available: http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=lua&lang2=python3

[8] S. Harihareswara, "New Lua templates bring faster, more flexible pages to your wiki," Mar 2013, (Accessed: Jan 2017). [Online]. Available: https://blog.wikimedia.org/2013/03/11/lua-templates-faster-more-flexible-pages/

[9] M. Tourne, "Pushing Nginx to its limit with Lua," Dec 2012, (Accessed: Jan 2017). [Online]. Available: https://blog.cloudflare.com/pushing-nginx-to-its-limit-with-lua/

[10] C. Somerville, "Rearchitecting GitHub Pages," May 2015, (Accessed: Jan 2017). [Online]. Available: https://githubengineering.com/rearchitecting-github-pages/

[11] N. Kolban, "Kolbans book on the ESP8266," *Texas, USA*, 2015.

[12] Project-NodeMCU, "Nodemcu – an open-source firmware based on esp8266 wifi-soc," *URL http://nodemcu.com/indexen.html*, 2014. [Online]. Available: http://nodemcu.com/

[13] Project-NodeLua, "Nodelua – the first open source lua based firmware that runs on esp8266," *URL https://nodelua.org*, 2014. [Online]. Available: https://nodelua.org/

[14] "Where Lua Is Used," (Accessed: Jan 2017). [Online]. Available: https://sites.google.com/site/marbux/home/where-lua-is-used

[15] "A curated list of static analysis tools, linters and code quality checkers for various programming languages," (Accessed: Jan 2017). [Online]. Available: https://github.com/mre/awesome-static-analysis

[16] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 437–448.

[17] "LuaCheck – A tool for linting and static analysis of Lua code," (Accessed: Jan 2017). [Online]. Available: https://github.com/mpeterv/luacheck

[18] "lualint – Lua linter performs luac-based static analysis of global variable usage in Lua source code," (Accessed: Jan 2017). [Online]. Available: https://github.com/philips/lualint/

[19] "lua-checker – Check Lua source code for various errors," (Accessed: Jan 2017). [Online]. Available: https://code.google.com/archive/p/lua-checker/

[20] "OWASP Top Ten Project," (Accessed: Jan 2017). [Online]. Available: https://www.owasp.org/index.php/OWASP_Top_Ten_Project

[21] "HP – Fortify On Demand," (Accessed: Jan 2017). [Online]. Available: https://saas.hpe.com/en-us/software/fortify-on-demand/capabilities

[22] "Checkmarx – Language Overview," (Accessed: Jan 2017). [Online]. Available: https://www.checkmarx.com/language-overviews/

[23] "Kiuwan – Supported Languages," (Accessed: Jan 2017). [Online]. Available: https://www.kiuwan.com/languages/

[24] "Syhunt – Cross-Site Scripting (XSS)," (Accessed: Jan 2017). [Online]. Available: http://www.syhunt.com/docwiki/index.php?n=Vulnerabilities.XSS

[25] "Syhunt – Supported Technologies and Languages," (Accessed: Jan 2017). [Online]. Available: http://www.syhunt.com/docwiki/index.php?n=SyhuntHybrid5.Technologies

[26] G. Nilson, K. Wills, J. Stuckman, and J. Purtilo, "Bugbox: A vulnerability corpus for php web applications." in *CSET*, 2013.

[27] OWASP, "OWASP WebGoat Project," (Accessed: Jan 2017). [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

[28] S. Dashevskyi, D. R. Dos Santos, F. Massacci, and A. Sabetta, "TestREx: a Testbed for Repeatable Exploits." in *CSET*, 2014.

[29] "Lua: uses," (Accessed: Jan 2017). [Online]. Available: https://www.lua.org/uses.html

[30] "lua-users wiki: Lua Uses," (Accessed: Jan 2017). [Online]. Available: http://lua-users.org/wiki/LuaUses

[31] "Lua-scriptable software," (Accessed: Jan 2017). [Online]. Available: http://en.wikipedia.org/wiki/Category:Lua-scriptable_software

[32] "Tengine – a Nginx and Lua based web server originated by Taobao, the largest e-commerce website in Asia," (Accessed: Jan 2017). [Online]. Available: http://tengine.taobao.org/

[33] "Lua for World of Warcraft," 2016, (Accessed: Jan 2017). [Online]. Available: http://wowwiki.wikia.com/wiki/Lua

[34] "Nmap Scripting Engine – Lua Base Language," (Accessed: Jan 2017). [Online]. Available: https://nmap.org/book/nse-language.html

[35] H. Kaplan, "Lua – The Wireshark Wiki," Jul 2015, (Accessed: Jan 2017). [Online]. Available: https://wiki.wireshark.org/Lua

[36] "Lua Web Application Security Vulnerabilities," May 2014, (Accessed: Jan 2017). [Online]. Available: http://www.syhunt.com/en/index.php?n=Articles.LuaVulnerabilities

[37] Scott Tenaglia, "Killing Mirai: Active defense against an IoT botnet (Part 1)," Oct 2016, (Accessed: Jan 2017). [Online]. Available: https://www.invincealabs.com/blog/2016/10/killing-mirai/

[38] Symantec Security Response, "Strider: Cyberespionage group turns eye of Sauron on targets," May 2012, (Accessed: Jan 2017). [Online]. Available: https://www.symantec.com/connect/blogs/flamer-highly-sophisticated-and-discreet-threat-targets-middle-east

[39] Laboratory of Cryptography and System Security (CrySyS Lab), "sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks," May 2012, (Accessed: Jan 2017). [Online]. Available: http://www.crysys.hu/skywiper/skywiper.pdf

[40] M. Marschalek, "EvilBunny: Malware Instrumented By Lua," Dec 2014, (Accessed: Jan 2017). [Online]. Available: https://www.cyphort.com/evilbunny-malware-instrumented-lua/

[41] Kaspersky Lab's Global Research/Analysis Team, "ProjectSauron: top level cyber-espionage platform covertly extracts encrypted government comms," Aug 2016, (Accessed: Jan 2017). [Online]. Available: https://securelist.com/analysis/publications/75533/faq-the-projectsauron-apt/

[42] Symantec Security Response, "Strider: Cyberespionage group turns eye of Sauron on targets," Aug 2016, (Accessed: Jan 2017). [Online]. Available: https://www.symantec.com/connect/blogs/strider-cyberespionage-group-turns-eye-sauron-targets

[43] MalwareMustDie, "MMD-0057-2016 - Linux/LuaBot - IoT botnet as service," Sep 2016, (Accessed: Jan 2017). [Online]. Available: http://blog.malwaremustdie.org/2016/09/mmd-0057-2016-new-elf-botnet-linuxluabot.html

[44] Symantec, "Linux.Luabot," Sep 2016, (Accessed: Jan 2017). [Online]. Available: https://www.symantec.com/security_response/writeup.jsp?docid=2016-090915-3236-99&tabid=2

[45] B. Rodrigues, "LuaBot: Malware targeting cable modems," Sep 2016, (Accessed: Jan 2017). [Online]. Available: https://w00tsec.blogspot.com/2016/09/luabot-malware-targeting-cable-modems.html

[46] R. Dobbins, "Mirai IoT botnet description and ddos attack mitigation," *Arbor Threat Intelligence*, vol. 28, 2016.

[47] A. Costin, "Security of CCTV and Video Surveillance Systems: Threats, Vulnerabilities, Attacks, and Mitigations," in *TrustED'16: International Workshop on Trustworthy Embedded Devices Proceedings*, 2016.

[48] "OpenWrt – Linux distribution for embedded devices." (Accessed: Jan 2017). [Online]. Available: https://openwrt.org/

[49] "OpenWrt – Table of Hardware," (Accessed: Jan 2017). [Online]. Available: https://wiki.openwrt.org/toh/start

[50] "LinkIt Smart 7688 Resources – FAQ," (Accessed: Jan 2017). [Online]. Available: https://docs.labs.mediatek.com/resource/linkit-smart-7688/en/faq

[51] "Linino – Internet of Everything," (Accessed: Jan 2017). [Online]. Available: https://www.linino.org

[52] "LuCI – OpenWrt Configuration Interface Source Code," (Accessed: Jan 2017). [Online]. Available: https://github.com/openwrt/luci

[53] "LuCI – Technical Reference Wiki," (Accessed: Jan 2017). [Online]. Available: https://wiki.openwrt.org/doc/techref/luci

[54] Wind River, "Intelligent Device Platform XT," Oct 2013, (Accessed: Jan 2017). [Online]. Available: https://www.windriver.com/products/product-notes/PN_IDPXT/PN_IDPXT.pdf

[55] "The LuaJIT Project." (Accessed: Jan 2017). [Online]. Available: http://luajit.org/luajit.html

[56] "eLua Project – Embedded power, driven by Lua." (Accessed: Jan 2017). [Online]. Available: http://www.eluaproject.net/

[57] "The llvm-lua Project." (Accessed: Jan 2017). [Online]. Available: https://github.com/Neopallium/llvm-lua.git

[58] "mod_lua – Apache HTTP Server Version 2.5." (Accessed: Jan 2017). [Online]. Available: https://httpd.apache.org/docs/trunk/mod/mod_lua.html

[59] "OpenResty – a dynamic web platform based on NGINX and LuaJIT." (Accessed: Jan 2017). [Online]. Available: http://openresty.org/

[60] "OpenResty – a dynamic web platform based on NGINX and LuaJIT." (Accessed: Jan 2017). [Online]. Available: https://github.com/openresty/openresty

[61] "lua-nginx-module – Embed the Power of Lua into NGINX HTTP servers." (Accessed: Jan 2017). [Online]. Available: https://github.com/openresty/lua-nginx-module

[62] "ngx_lua," (Accessed: Jan 2017). [Online]. Available: https://launchpad.net/ngx-lua

[63] KeplerProject, "CGILua – Building Web Scripts with Lua." (Accessed: Jan 2017). [Online]. Available: https://keplerproject.github.io/cgilua/

[64] ——, "WSAPI – Lua Web Server API." (Accessed: Jan 2017). [Online]. Available: https://keplerproject.github.io/wsapi/

[65] D. Scott and R. Sharp, "Abstracting application-level web security," in *Proceedings of the 11th international conference on World Wide Web*. ACM, 2002, pp. 396–407.

[66] M. Hashimoto, *Vagrant: Up and Running*. " O'Reilly Media, Inc.", 2013.

[67] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[68] "Lua – Found 10,132 repository results," (Accessed: Jan 2017). [Online]. Available: https://github.com/search?l=Lua&q=lua&type=Repositories&utf8=%E2%9C%93

[69] "Lua – Trending in open source," (Accessed: Jan 2017). [Online]. Available: https://github.com/trending/lua?since=monthly

[70] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[71] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 63–72.

[72] A. Hemel, "BAT – Binary Analysis Toolkit," (Accessed: Jan 2017). [Online]. Available: http://binaryanalysis.org/

[73] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 989–1003.

[74] J. Dahse, "RIPS – A static source code analyser for vulnerabilities in PHP scripts." in *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum*, 2010.

[75] F. Yamaguchi, "Joern – An Intelligent Code Analysis Platform for C/C++," (Accessed: Jan 2017). [Online]. Available: http://www.mlsec.org/joern/

[76] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 359–368.

[77] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 590–604.

[78] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 797–812.

[79] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *2008 IEEE Symposium on Security and Privacy (SP 2008)*. IEEE, 2008, pp. 387–401.

[80] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1977, pp. 238–252.

[81] K.-G. Doh, H. Kim, and D. A. Schmidt, "Abstract parsing: static analysis of dynamically generated string output using lr-parsing technology," in *International Static Analysis Symposium*. Springer, 2009, pp. 256–272.

[82] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 40–52.

[83] G. A. Di Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana, "Identifying cross site scripting vulnerabilities in web applications," in *Web Site Evolution, Sixth IEEE International Workshop on (WSE'04)*. IEEE, 2004, pp. 71–80.

[84] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 6–pp.

[85] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages." in *USENIX Security*, vol. 6, 2006, pp. 179–192.

[86] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 171–180.

[87] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis." in *Usenix Security*, vol. 2013, 2005.

[88] P. Biggar and D. Gregg, "Static analysis of dynamic scripting languages."

[89] J. Dahse and T. Holz, "Simulation of Built-in PHP Features for Precise Static Code Analysis." in *NDSS*, 2014.

[90] OWASP, "WebGoat – a deliberately insecure Java web application." (Accessed: Jan 2017). [Online]. Available: https://github.com/WebGoat/WebGoat

[91] "BugBox Homepage – A Vulnerability Corpus for Web Applications." (Accessed: Jan 2017). [Online]. Available: https://bugbox.cs.umd.edu/

[92] "BugBox on Github – A Vulnerability Corpus for Web Applications." (Accessed: Jan 2017). [Online]. Available: https://github.com/UMD-SEAM/bugbox

[93] S. Dashevskyi, " TestREx – a testbed for repeatable web application security experiments." (Accessed: Jan 2017). [Online]. Available: https://github.com/standash/TestREx