MAQAO Lua API

High level programming using MAQAO internals

Version 1.0

www.maqao.org

Contents

1 Introduction

MAQAO is a toolchain that provides a set of tools targeting different issues at different levels by means of a top/down methodology. The coarse grain tools pinpoint high level bottlenecks such as finding most time consuming functions and loops (hotspots). Fine grain tools target specific issues such as the characterization of the memory behavior of a loop or the quality of the code generated by the compiler for a specific part of an application.

These tools are built on top of the MAQAO framework (Figure 1) which is a set of reusable building blocks. We can mainly distinguish four layers:

- the binary manipulation layer which aim is to provide a mean to perform operations at binary levels using an abstracted layer
- the structuration and analysis layer which transforms the assembly representation into high level objects such as instructions, basic blocks, loops and functions.
- The scripting language layer
- the high level Lua modules.

High level module can also invoke each other. For instance the Profiler module uses MIL scripts when using the instrumentation measurement method.

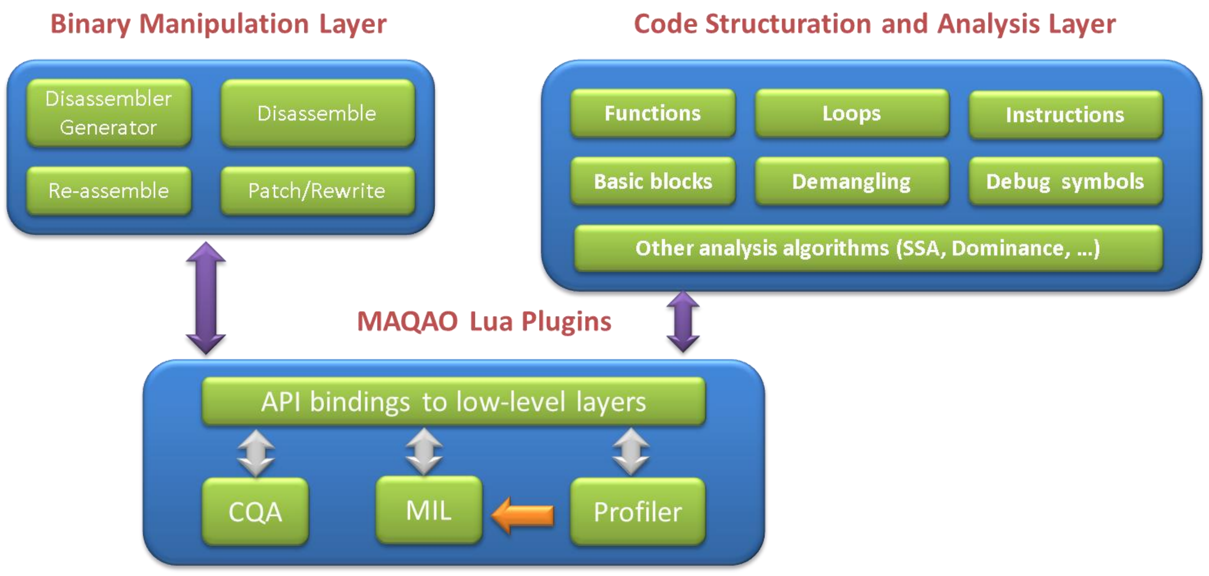

Figure 1: MAQAO Framework overview

2 MAQAO Lua API

Lua Modules are based on the Lua scripting language and the MAQAO Lua API which exposes methods to access the binary manipulation and structuration and analysis layers.

2.1 Lua

The authors' description of the language remains the best. The following test presents it.

2.1.1 Overview

Lua is a powerful, fast, lightweight, embeddable scripting language.

Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

Lua has been used in many industrial applications (e.g., Adobe's Photoshop Lightroom), with an emphasis on embedded systems (e.g., the Ginga middleware for digital TV in Brazil) and games (e.g., World of Warcraft and Angry Birds). Lua is currently the leading scripting language in games. Lua has a solid reference manual and there are several books about it. Several versions of Lua have been released and used in real applications since its creation in 1993.

2.1.2 Lua is fast

Lua has a deserved reputation for performance. To claim to be "as fast as Lua" is an aspiration of other scripting languages. Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages. Lua is fast not only in fine-tuned benchmark programs, but in real life too. Substantial fractions of large applications have been written in Lua.

If you need even more speed, try LuaJIT, an independent implementation of Lua using a just-in-time compiler.

2.1.3 Lua is portable

Lua is distributed in a small package and builds out-of-the-box in all platforms that have a standard C compiler. Lua runs on all flavors of Unix and Windows, on mobile devices (running Android, iOS, BREW, Symbian, Windows Phone), on embedded microprocessors (such as ARM and Rabbit, for applications like Lego MindStorms), on IBM mainframes, etc.

For specific reasons why Lua is a good choice also for constrained devices, read this summary by Mike Pall. See also a poster created by Timm MÃ¼ller.

2.1.4 Lua is embeddable

Lua is a fast language engine with small footprint that you can embed easily into your application. Lua has a simple and well documented API that allows strong integration with code written in other languages. It is easy to extend Lua with libraries written in other languages. It is also easy to extend programs written in other languages with Lua. Lua has been used to extend programs written not only in C and C++, but also in Java, C#, Smalltalk, Fortran, Ada, Erlang, and even in other scripting languages, such as Perl and Ruby.

2.1.5 Lua is powerful (but simple)

A fundamental concept in the design of Lua is to provide meta-mechanisms for implementing features, instead of providing a host of features directly in the language. For example, although Lua is not a pure object-oriented language, it does provide meta-mechanisms for implementing classes and inheritance. Lua's meta-mechanisms bring an economy of concepts and keep the language small, while allowing the semantics to be extended in unconventional ways.

2.1.6 Lua is small

Adding Lua to an application does not bloat it. The tarball for Lua 5.2.2, which contains source code and documentation, takes 246K compressed and 960K uncompressed. The source contains around 20000 lines of C. Under Linux, the Lua interpreter built with all standard Lua libraries takes 182K and the Lua library takes 243K.

2.1.7 Lua is free

Lua is free open-source software, distributed under a very liberal license (the well-known MIT license). It may be used for any purpose, including commercial purposes, at absolutely no cost. Just download it and use it.

2.2 Lua in MAQAO

We have embedded Lua in MAQAO. It can execute a Lua script as the default Lua interpreter does.

```
maqao my_lua_script.lua
```

The main goal is to provide application developers with a simple fast prototyping but powerful environment. The MAQAO Lua API extends the classical LUA API with a set of objects and associated methods in order to be able to perform MAQAO specific actions like manipulating a binary file and/or performing static analyses. Notice that it is also possible to perform dynamic analyses by using the MAQAO Instrumentation Language (MIL) which is covered in another tutorial.

Along with the access to MAQAO internals, it is also possible to use existing MAQAO modules written in Lua. This will be covered in an upcoming tutorial.

When performance matters, it is possible to transfer the performance-sensitive part to a C library that can be loaded from your Lua's module. For more information please refer to the Lua C API in Lua's manual.

2.3 MAQAO objects

As mentioned before the MAQAO Lua API exposes a set of objects and their methods. The available objects are the following:

- instructions: assembly instructions with all their details
- basic blocks: a set of contiguous instructions bounded either by a branch, a return, a call or a next instruction being the target of a branch.
- loops: natural loops found in the CFG
- functions: based on the binary format information. Functions are contiguous in the text section of a binary

Figure 2 presents the hierarchy between these objects.

Appendix A lists all the objects and associated methods provided by the MAQAO Lua API.

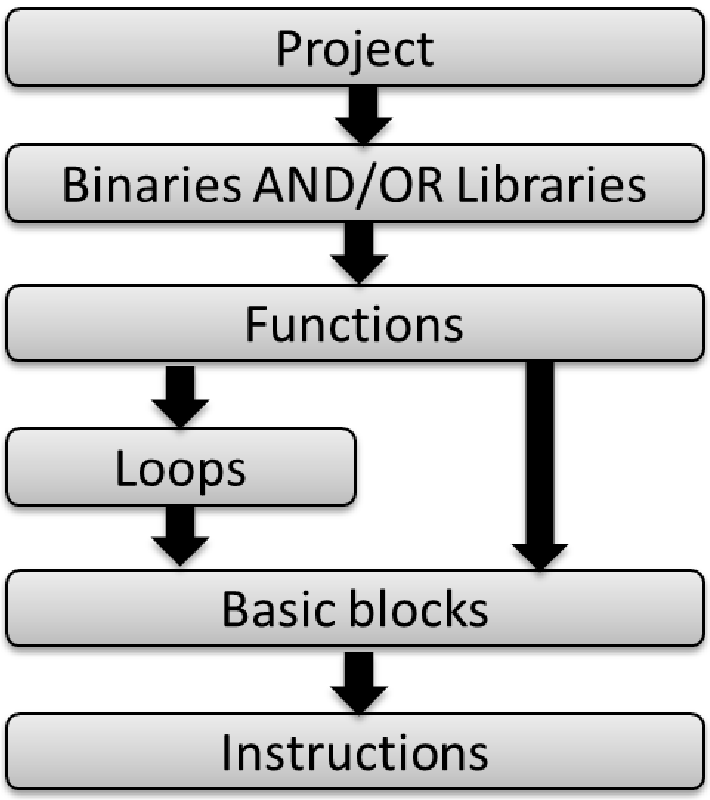

Figure 2: MAQAO objects hierarchy

3 Examples

This section presents some examples of what can be achieved in a very few lines.

3.1 Listing functions, loops, basic blocks and instructions

The MAQAO script below shows how to display the following MAQAO objects from a given binary:

- function (name)
- assembly loop (id)
- basic block (id)
- instruction (address)

N.B.: identifiers mentioned above are unique identifiers only relevant to MAQAO.

```
--Create a project and load a given binary
local proj = project.new("Project Name");
--arg[1] contains the first command line parameter
local bin = proj:load(arg[2], 0);

--Go through the abstract objects hierarchy
--and display their name/id
for f in bin:functions() do
  print("Function "..f:get_name());
  for l in f:loops() do
    print(" Loop id "..l:get_id());
    for b in l:blocks() do
      print("  Block id "..b:get_id());
      for i in b:instructions() do
          print(i);
      end
    end
  end
end
```

This script iteratively displays for every function in the binary its name. Then for every function it displays every assembly loop identifier. The same happens for basic blocks in loops and instructions in basic blocks. We recognize the hierarchy described in Figure 2.

3.2 Listing load instructions

The following script all the load instructions for each function of a given binary.

```lua
--Create a project and load a given binary
local proj = project.new("Load instructions");
--arg[2] contains the first command line parameter
local bin = proj:load(arg[2], 0);

--Go through the abstract objects hierarchy
--and filter only load memory instructions
for f in bin:functions() do
  print("\nFunction "..f:get_name());
  for b in f:blocks() do
    for i in b:instructions() do
      if(i:is_load()) then
        -- display instruction and then the first memory operand
        local memory_operand = i:get_first_mem_oprnd();
        print(i,memory_operand);
      end
    end
  end
end
```

Note that here we can bypass the loop objects by directly considering the basic blocks of a functions (see Figure 2).

3.3 Objdump-like disassembler

This example demonstrated the ease with which it is possible to create in a few lines an objdump-like disassembler. It can then be modified as desired

```lua
--Create a project and load a given binary
local proj = project.new("Load instructions");
--arg[2] contains the first command line parameter
local bin = proj:load(arg[2], 0);

for fct in a:functions() do
  print("<"..fct:get_name()..">:");
  for b in fct:blocks() do
     for ins in b:instructions() do
        print(string.format("%x",ins:get_address()).."\t"
           ..string.format("%8s",ins:get_coding()).."\t"
           ..ins:tostring());
     end
  end
end
```

4 One step further: Modules

After having written a couple of scripts validating a proof-of-concept tool tailored to specific needs, one might want to invest more time transforming these scripts into a module providing means to give added structure to the tool (handle properly parameters, help, etc…).

To learn how to create modules, please refer to the corresponding tutorial: “Creating a module”.

5 Appendix A

The detailed function prototypes of the current MAQAO Lua API are available in the tutorial’s associated download MAQAO.LuaAPI.tar.bz2. The tarball contains a luadoc browsable documentation.