解析android手游lua脚本的加密与解密（番外篇之反编译的对抗）

转载

一生磨一剑 于 2019-07-25 18:24:56 发布 1206 收藏 3

转载自https://blog.csdn.net/xjpdf10/article/details/82219080

前言

去年在看雪论坛写了一篇《浅析android手游lua脚本的加密与解密》的精华文章，今年写一篇番外篇，将一些lua反编译对抗的内容整合一起，并以3个实例作为说明（包括2018腾讯游戏竞赛和梦幻西游手游相关的补充），文章开头还增加了相关工作，方便大家学习lua逆向时使用。本文由3篇文章整合成1篇，所以内容上面有点多，有兴趣的朋友需要点耐心，当然也可以跳着看。最后，请大佬们不吝赐教。最最后，大家有问题也欢迎留言，一起交流学习。

相关工作

为了能让一些同学更好的学习lua的逆向，我把收集的一些资料组合成一篇lua加解密的相关工作给大家参考。看这节内容之前还是需要一些lua的基础知识，这里推荐云风大佬的《Lua源码欣赏》[19]，建议结合搜索引擎学习之。

文章分2部分介绍，第1部分介绍lua加解密的相关文章介绍，第2部分介绍lua的相关工具。

文章介绍

这一节介绍了互联网上对lua的各种相关文章，包括lua的加解密如文件格式的解析、基于lua的游戏和比赛的介绍、lua的hook技术等。

1. lua加解密入门：

非虫大佬[1-4] 写了4篇关于luac和luajit文件格式和字节码的相关文章，并开源了010Editor的解析luac和luajit的模板代码。Ganlv 同学[7] 在吾爱破解写了7篇关于lua加解密的系列教程。腾讯gslab[9] 写了一篇关于lua游戏逆向的入门介绍，这是一篇比较早的lua游戏解密的文章。lNightElf 同学[10] 写了一篇关于lua脚本反编译入门的文章。

2. 基于lua的手游：

lua不仅能用于端游戏，也能用于手游，而且由于手游的火热，带动了lua逆向相关分析文章的分享。wmsuper 同学[11] 在android平台下解密了腾讯游戏开心消消乐的lua脚本，后续可以通过修改lua脚本达到作弊的目的。Unity 同学[8] 通过hook的方法解密和修改lua手游《放置江湖》的流程，达到修改游戏奖励的目的。littleNA 同学[12] 通过3种方式解密了3个手游的lua脚本，并且修复了梦幻手游lua opcode的顺序。

3. 基于lua的比赛：

随着国内CTF的发展，lua技术也运用到了比赛中。看雪ctf2016第2题[13]、2017第15题[14]和腾讯游戏安全2018决赛第2题[15]都使用了lua引擎作为载体的CrackMe比赛，其中看雪2016将算法验证用lua代码实现并编译成luac，最后还修改了luac的文件头，使得反编译工具报错；看雪2017的题使用壳和大量的混淆，最后一步是luajit的简单异或运算；腾讯2018使用的lua技术更加深入，进阶版更是修改了lua的opcode顺序，并使用lua编写了一个虚拟机。以上3题的writeup网上都可以搜索到，有兴趣的朋友可以练练手，加深印象。

4. lua hooking：

Hook是修改软件流程的常用手段，lua中也存在hook技术。曾半仙 同学[9] 在看雪发布了一种通过hook lua字节码达到修改游戏逻辑的方法，并发布了一个lua汇编引擎。Nikc Cano[5] 的blog写了一篇关于Hooking luajit的文章，興趣使然的小胃 同学[6] 对该篇文章进行了翻译。

工具介绍

逆向解密lua和luajit游戏都有相关的工具，这一节将对一些主流的工具进行介绍。

1. lua相关：

luadec [16]：这是一个用c语言结合lua引擎源码写的开源lua反编译器，解析整个lua字节码文件并尽可能的还原为源码。当然，由于还原的是高级语言，所以兼容性一般，当反编译大量文件时肯定会遇到bug，这时就需要自己手动修复bug；并且很容易被针对造成反编译失败。目前支持的版本有lua5.1，5.2和5.3。

chunkspy：一款非常有用的lua分析工具，本身就是lua语言所写。它解析了整个lua字节文件，由于其输出的是lua的汇编形式，所以兼容性非常高，也造成了一定的阅读障碍。chunkspy 不仅可以解析luac文件，它还包括了一个交互式的命令，可以将输入的lua脚本转换成lua字节码汇编的形式，这对学习lua字节码非常有帮助。luadec工具中集成了这个脚本，目前支持的版本也是有lua5.1，5.2和5.3。

unluac：这也是一个开源的lua反编译器，java语言所写，相比luadec 工具兼容性更低,。一般很少使用，只支持lua5.1，当上面工具都失效时可以尝试。

2. luajit相关：

luajit-decomp[17]：github开源的一款luajit反编译工具，使用au3语言编写。先通过luajit原生的exe文件将luajit字节码文件转换成汇编，然后该工具再将luajit汇编转换成lua语言。由于反汇编后的luajit字节码缺少很多信息，如变量名、函数名等，造成反编译后的结果读起来比较隐晦，类似于IDA的F5。但是兼容性超好，只要能够反汇编就能够反编译，所以使用时需要替换对应版本的luajit引擎（满足反汇编的需求）。目前是支持所有的luajit版本。

ljd[18]：也是github开源的一款luajit反编译工具，使用python编写，与luajit-decomp 反编译luajit汇编的方式不同，其从头解析了整个luajit文件，能够获取更多的信息，还原的程度更高，但是由于精度更高，所以兼容性也会弱一点。查看该项目的fork可以获取更多的其他兼容版本，目前支持的版本有luajit2.0、luajit2.1等。

反编译对抗

众所周知，反汇编/反编译 工具在逆向人员工作中第一步被使用，其地位非常之高，而对于软件保护者来说，如何对抗 反汇编/反编译 就显得尤为重要。例如，动态调试中对OD的的检测、内核调试对windbg的破坏、加壳加花对IDA静态分析的阻碍、apktool的bug导致对修改后的apk反编译失败、修改PE头导致OD无法识别、修改 .Net dll中的区段导致ILspy工具失效等等例子，都说明对抗反编译工具是很常用的一种软件保护手段。当然，lua的反编译工具也面临这个问题。处理这样的问题无非就几种思路：

用调试器调试反编译工具为何解析错误，排查原因。

用调试器调试原引擎是如何解析文件的。

用文件格式解析工具解析文件，看哪个点解析出错。

下面将以3个例子来实战lua反编译是如何对抗与修复。

例子1：一个简单的问题

这是在看雪论坛看到的一个问题，问题是由于游戏（可能是征途手游）将lua字符串的长度int32修改为int64，导致反编译失败的一个例子，内容较为简单，修复方法请看帖子中本人的回答，地址：https://bbs.pediy.com/thread-217033.htm

例子2：2018腾讯游戏安全竞赛

这一节以2018腾讯游戏安全竞赛决赛第二题进阶版第1关的题目为例子，主要是讲一下如何修复当lua的opcode被修改的情况，以及如何修复该题对抗lua反编译的问题。

opcode问题及其修复

修复opcode的目的是 当输入题目的luac文件，反汇编工具Chunkspy和反编译工具luadec能够输出正确的结果。

首先，我们在ida中分析lua引擎tmgs.dll文件，然后定位到luaV_execute函数（搜索字符串“ 'for' limit must be a number ”），发现switch下的case的参数（lua的opcode）是乱序的，到这里我们就能够确认，该题的lua虚拟机opcode被修改了。

接着，我们进行修复操作。一种很耗时的办法就是一个一个opcode还原，分析每一个case下面的代码然后找出对应opcode的顺序。但是这一题我们不用这么麻烦，通过对比分析我们发现普通版的题目并没有修改opcode：

普通版lua引擎的luaV_execute函数

```
495     switch ( v8 & 0x3F )
496     {
497       default:
498         continue;
499       case 0u:
500         *(_OWORD *)v19 = *(_OWORD *)(v4 + 16 * (v8 >> 23));
501         continue;
502       case 1u:
503         *(_OWORD *)v19 = *(_OWORD *)(v319 + 16 * (v8 >> 14));
504         continue;
505       case 2u:
506         v6 = v260;
507         v20 = *(unsigned int **)(v260 + 20);
508         v21 = (_OWORD *)(v319 + 16 * (*v20 >> 6));
509         *(_DWORD *)(v260 + 20) = v20 + 1;
510         *(_OWORD *)v19 = *v21;
511         goto LABEL_37;
512       case 3u:
513         *(_DWORD *)(v19 + 8) = 1;
514         v22 = v8 >> 23;
515         v9 = (v8 & 0x7FC000) == 0;
516         v6 = v260;
517         *(_DWORD *)v19 = v22;
518         if ( !v9 )
519           *(_DWORD *)(v260 + 20) += 4;
520         goto LABEL_8;
521       case 4u:
522         v23 = v8 >> 23;
523         v24 = (_DWORD *)(v19 + 8);
524         do
525         {
526           v25 = v23;
527           *v24 = 0;
528           --v23;
529           v24 += 4;
530         }
531         while ( v25 );
532         goto LABEL_6;
```

进阶版lua引擎的luaV_execute函数

```
450     switch ( BYTE4(v7) & 0x3F )
451     {
452       default:
453         goto LABEL_7;
454       case 6:
455       case 7:
456       case 0x16:
457       case 0x1B:
458         v19 = *(_OWORD *)(v6 + 16 * (HIDWORD(v7) >> 23));
459         goto LABEL_6;
460       case 0x22:
461       case 0x28:
462       case 0x29:
463       case 0x3C:
464         v19 = *(_OWORD *)(v298 + 16 * (HIDWORD(v7) >> 14));
465         goto LABEL_6;
466       case 0x3E:
467         v20 = *(unsigned int **)(v246 + 20);
468         v21 = (__int128 *)(v298 + 16 * (*v20 >> 6));
469         *(_DWORD *)(v246 + 20) = v20 + 1;
470         v19 = *v21;
471         goto LABEL_5;
472       case 0x3B:
473         *(_DWORD *)(v18 + 8) = 1;
474         *(_DWORD *)v18 = HIDWORD(v7) >> 23;
475         v5 = v246;
476         if ( HIDWORD(v7) & 0x7FC000 )
477           *(_DWORD *)(v246 + 20) += 4;
478         continue;
479       case 0x12:
480         v22 = HIDWORD(v7) >> 23;
481         v23 = (_DWORD *)(v18 + 8);
482         do
483         {
484           v24 = v22;
485           *v23 = 0;
486           --v22;
487           v23 += 4;
488         }
489         while ( v24 );
490         goto LABEL_7;
```

通过对比代码，发现对应未修改的case 0

对应case 1

对应case 2

case 3

case 4

观察发现，进阶版的题目只是修改了每个case的数值或者多个值映射到同一个opcode，但是没有打乱case里的代码（也就是说，虚拟机解析opcode代码的顺序没有变，只是修改了对应的数值，这跟梦幻手游的打乱opcode的方法不同）。由于lua5.3只使用到0x2D的opcode，而一个opcode长度为6位（0x3F），该题就将剩余的没有使用的字节映射到同一个opcode下，修复时只需要反过来操作就可以了。分析到这里，我们的修复方案就出来了：

通过ida分别导出2个版本的 luaV_execute 的文本

通过python脚本提取opcode的修复表

在工具（Chunkspy和luadec）初始化lua文件后，用修复表将opcode替换

测试运行，修复其他bug

第一步直接IDA手动导出: File --> Produce file --> Create LST File ；第二步使用python分析，代码如下：

```
# -*- coding: utf-8 -*-# 通过扫码IDA导出的文本文件，获取lua字节码的opcode顺序
def get_opcode(filepath):    f = open(filepath)
    lines = f.readlines()
    opcodes = []

    # 循环扫码文件的每一行    for i in range(len(lines)):
        line = lines[i]
        if line.find('case') != -1:
            line = line.replace('case', '')
            line = line.replace(' ', '')
            line = line.replace('\n','')
            line = line.replace('u:', '')

            # 如果上一行也是case，那么这2个case对应同一个opcode            if lines[i-1].find('case') != -1:
                opcode = opcodes[-1]
                opcode.append(line)
            else:
                opcode = []
                opcode.append(line)
                opcodes.append(opcode)
    f.close()
    return opcodes

o1 = get_opcode(u'基础版opcode.txt')
o2 = get_opcode(u'进阶版opcode.txt')# 还原for i in range(len(o1)):
    print '基础版：',o1[i],'\t进阶版：',o2[i]# 映射opcode获取修复表op_tbl = [-
1 for i in range(64)]for i in range(len(o1)):
    o1opcode = o1[i][0]
    o1opcode = o1opcode.replace('0x','')

    for o2opcode in o2[i]:
        o2opcode = o2opcode.replace('0x','')
        op_tbl[int(o2opcode,16)] = int(o1opcode,16)print '修复表：',op_tbl
```

运行结果：

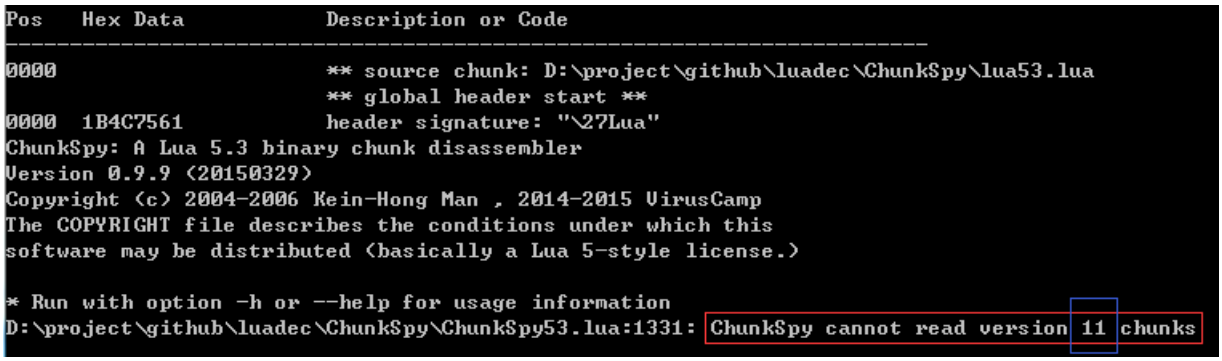
```
基础版： ['0']        进阶版： ['6', '7', '0x16', '0x1B']
基础版： ['1']        进阶版： ['0x22', '0x28', '0x29', '0x3C']
基础版： ['2']        进阶版： ['0x3E']
基础版： ['3']        进阶版： ['0x3B']
基础版： ['4']        进阶版： ['0x12']
基础版： ['5']        进阶版： ['8', '0x11', '0x17', '0x36']
基础版： ['6']        进阶版： ['2']
基础版： ['7']        进阶版： ['0xD']
基础版： ['8']        进阶版： ['0x1A']
基础版： ['9']        进阶版： ['1']
基础版： ['0xA']      进阶版： ['0x1D']
基础版： ['0xB']      进阶版： ['0x1F']
基础版： ['0xC']      进阶版： ['0xE']
基础版： ['0xD']      进阶版： ['0x31']
基础版： ['0xE']      进阶版： ['0x2F']
基础版： ['0xF']      进阶版： ['0x1E']
基础版： ['0x12']     进阶版： ['0x13']
基础版： ['0x14']     进阶版： ['0x2B']
基础版： ['0x15']     进阶版： ['0x1C']
基础版： ['0x16']     进阶版： ['0x2D']
基础版： ['0x17']     进阶版： ['0x19']
基础版： ['0x18']     进阶版： ['0x3F']
基础版： ['0x10']     进阶版： ['0x15']
基础版： ['0x13']     进阶版： ['0x24']
基础版： ['0x11']     进阶版： ['0x3A']
基础版： ['0x19']     进阶版： ['0x18']
基础版： ['0x1A']     进阶版： ['0x33']
基础版： ['0x1B']     进阶版： ['0xF']
基础版： ['0x1C']     进阶版： ['0x34']
基础版： ['0x1D']     进阶版： ['0x20']
基础版： ['0x1E']     进阶版： ['5', '9', '0xA', '0x25']
基础版： ['0x1F']     进阶版： ['0x30']
基础版： ['0x20']     进阶版： ['0x26']
基础版： ['0x21']     进阶版： ['0x35']
基础版： ['0x22']     进阶版： ['0x38']
基础版： ['0x23']     进阶版： ['0x2A']
基础版： ['0x24']     进阶版： ['0x23', '0x37', '0x39', '0x3D']
基础版： ['0x25']     进阶版： ['0x27']
基础版： ['0x27']     进阶版： ['0x2C']
基础版： ['0x28']     进阶版： ['0x32']
基础版： ['0x29']     进阶版： ['0x21']
基础版： ['0x2A']     进阶版： ['3']
基础版： ['0x2B']     进阶版： ['0xC']
基础版： ['0x2C']     进阶版： ['0x2E']
基础版： ['0x2D']     进阶版： ['0x14']
基础版： ['0x26']     进阶版： ['4']
修复表： [-1, 9, 6, 42, 38, 30, 0, 0, 5, 30, 30, -1, 43, 7, 12, 27, -
1, 5, 4, 18, 45, 16, 0, 5, 25, 23, 8, 0, 21, 10, 15, 11, 29, 41, 1, 36, 19, 30, 32, 37, 1, 1, 35, 20, 39, 22
, 44, 14, 31, 13, 40, 26, 28, 33, 5, 36, 34, 36, 17, 3, 1, 36, 2, 24]
```

注意了，这里有几个opcode是没有对应关系的（默认是-1），跟踪代码发现，其实这些opcode的功能相当于nop操作，而原本lua是不存在nop的，我们只需在修复的过程中跳过这个字节码即可。

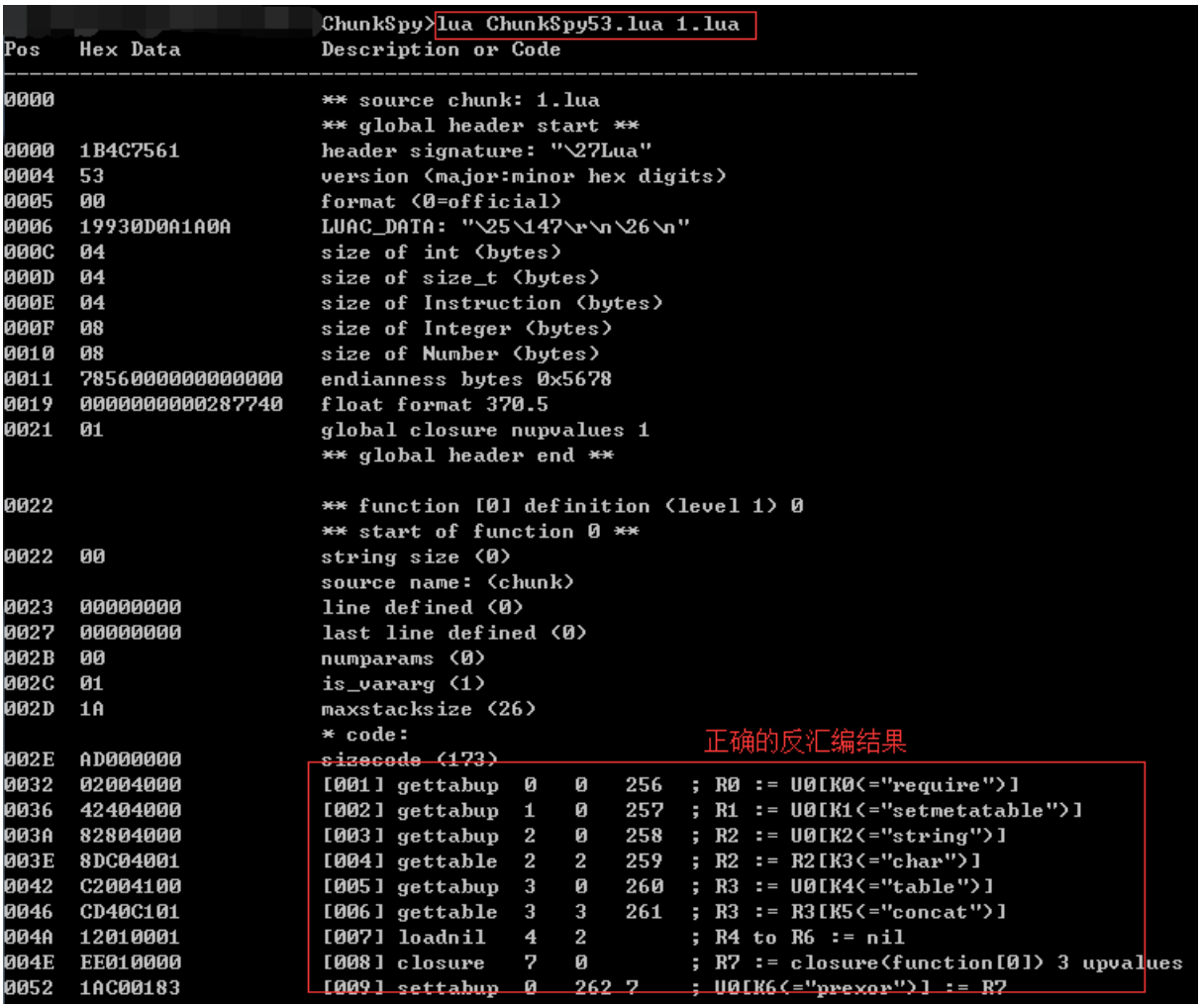
最后将获取的修复表替换到工具中，Chunspy修复点在DecodeInst函数中，修改结果如下：

```
function DecodeInst(code, iValues)  local iSeq, iMask = config.iABC, config.mABC
  local cValue, cBits, cPos = 0, 0, 1  -- decode an instruction  for i = 1, #iSeq do    -
- if need more bits, suck in a byte at a time    while cBits < iSeq[i] do      cValue = string.byte(code, cP
os) * (1 << cBits) + cValue
      cPos = cPos + 1; cBits = cBits + 8    end    -
- extract and set an instruction field    iValues[config.nABC[ i ]] = cValue % iMask[i]
    cValue = cValue // iMask[i]
    cBits = cBits - iSeq[i]
  end  -- add by littleNA  local optbl = { -1, 9, 6, 42, 38, 30, 0, 0, 5, 30, 30, -1, 43, 7, 12, 27, -
1, 5, 4, 18, 45, 16, 0, 5, 25, 23, 8, 0, 21, 10, 15, 11, 29, 41, 1, 36, 19, 30, 32, 37, 1, 1, 35, 20, 39, 22
, 44, 14, 31, 13, 40, 26, 28, 33, 5, 36, 34, 36, 17, 3, 1, 36, 2, 24 }
  iValues.OP = optbl[iValues.OP+1]  -- 注意，lua的下标是从1开始的数起的  -
- add by littleNA end  iValues.opname = config.opnames[iValues.OP]   -
- get mnemonic  iValues.opmode = config.opmode[iValues.OP]-- add by littleNA  if iValues.OP == -
1 then    iValues.opname = "Nop"    iValues.opmode = iABx
  end  -- add by littleNA endif iValues.opmode == iABx then                -
- set Bx or sBx    iValues.Bx = iValues.B * iMask[3] + iValues.C
  elseif iValues.opmode == iAsBx then    iValues.sBx = iValues.B * iMask[3] + iValues.C - config.MAXARG_sBx
  elseif iValues.opmode == iAx then    iValues.Ax = iValues.B * iMask[3] * iMask[2] + iValues.C * iMask[2] +
 iValues.A
  end  return iValuesend
```

测试发现出错了，出错结果：



从出错的结果可以看出是luac文件的版本号有错误，这里无法识别lua 11的版本其实是题目故意设计让工具识别错误，我们将文件的第4个字节（lua版本号）11修改成53就可以了。正确结果：



luadec修复点在ldo.c文件的f_parser函数，并且增加一个RepairOpcode函数，修复如下：

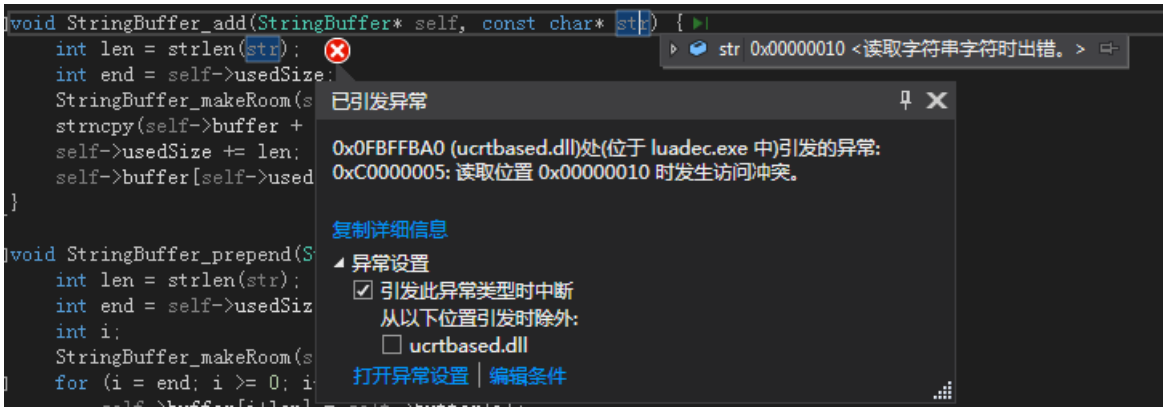
```
// add by littleNAvoid RepairOpcode(Proto* f){
    // opcode 替换表    char optbl[] = { -1, 9, 6, 42, 38, 30, 0, 0, 5, 30, 30, -1, 43, 7, 12, 27, -
1, 5, 4, 18, 45, 16, 0, 5, 25, 23, 8, 0, 21, 10, 15, 11, 29, 41, 1, 36, 19, 30, 32, 37, 1, 1, 35, 20, 39, 22
, 44, 14, 31, 13, 40, 26, 28, 33, 5, 36, 34, 36, 17, 3, 1, 36, 2, 24 };
    for (int i = 0; i < f->sizecode; i++)
    {
        Instruction code = f->code[i];
        OpCode o = GET_OPCODE(code);
        SET_OPCODE(code, optbl[o]);

        f->code[i] = code;
    }

    for (int i = 0; i < f->sizep; i++)
    {// 处理子函数        RepairOpcode(f->p[i]);
    }
}// add by littleNA endstatic void f_parser (lua_State *L, void *ud) {
  LClosure *cl;
  struct SParser *p = cast(struct SParser *, ud);
  int c = zgetc(p->z);  /* read first character */  if (c == LUA_SIGNATURE[0]) {
    checkmode(L, p->mode, "binary");
    cl = luaU_undump(L, p->z, p->name);

    // add by littleNA    Proto *f = cl->p;
    RepairOpcode(f);
    // add by littleNA end  }
  else {
    checkmode(L, p->mode, "text");
    cl = luaY_parser(L, p->z, &p->buff, &p->dyd, p->name, c);
  }
  lua_assert(cl->nupvalues == cl->p->sizeupvalues);
  luaF_initupvals(L, cl);
}
```

运行一下，发现出错了，并且停留在StringBuffer_add函数中，其中str指向错误的地方，导致字符串读取出错：



到这里我们修复了opcode，并且Chunkspy顺利反汇编，但是luadec的反编译还是有问题，我们在下一节分析。

反编译问题及其修复

看了几个大佬的writeup，发现他们都没有修复这个问题，解题过程中都是直接分析的是lua汇编代码。我们看看出错的原因，查看vs的调用堆栈：

发现上一层函数是listUpvalues函数，也就是说luadec在解析upvalues时出错了，深入分析发现其实是由于文件中的upvalue变量名被抹掉了，导致解析出错，我们只需要在ProcessCode函数（decompile.c文件）调用listUpvalues函数前，增加临时的upvalue命名就可以了，修改代码如下：

```
char* ProcessCode(Proto* f, int indent, int func_checking, char* funcnumstr)
{
...
	// make function comment	StringBuffer_printf(str, "-- function num : %s", funcnumstr);
	if (NUPS(f) > 0) {
		// add by littleNA		for (i = 0; i<f->sizeupvalues; i++) {
			char tmp[10];
			sprintf(tmp, "up_%d", i);
			f->upvalues[i].name = luaS_new(f->L, tmp);
		}
		// add by littleNA end		StringBuffer_add(str, " , upvalues : ");
		listUpvalues(f, str);
	}
...
}
```

最后完美运行luadec，反编译成功。

例子3：梦幻西游手游

这一节是去年学习破解梦幻西游手游lua代码时记录的一些问题，今天将其整理并共享出来，所以不一定适合现在版本的梦幻手游，大家还是以参考为目的呗。

当时反编译梦幻西游手游时遇到的问题大约有12个，修改完基本上可以完美复现lua源码，这里用的luadec5.1版本。

修复一

问题1： 由于梦幻手游lua的opcode是被修改过的，之前的解决方案是找到梦幻的opcode，替换掉反编译工具的原opcode，并且修改opmode，再进行反编译。问题是部分测试的结果是可以的，但是当对整个手游的luac字节码反编译时，会出现各种错误，原因是luadec5.1 在很多地方都默认了opcode的顺序，并进行了特殊处理，所以需要找到这些特殊处理的地方一一修改。不过这样很麻烦，从而想到另外一种方式，不修改原来的opcode和opmode，而是在luadec解析到字节码的时候，将opcode还原成原来的opcode。

解决1： 定位到解析code的位置在 lundump.c --> LoadFunction --> LoadCode （位置不唯一，可以看上一节腾讯比赛的修复），当执行完LoadCode函数的时候，f变量则指向了code的结构，在这之后执行自己写的函数ConvertCode函数，如下：

```
// add by littleNAvoid ConvertCode(Proto *f){
        int pnOpTbl[] = { 3,13,18,36,27,10,20,25,34,2,32,15,30,16,31,9,26,24,29,1,6,28,4,17,33,0,7,11,5,14,8
,19,35,12,21,22,23,37 };
        for (int pc = 0; pc < f->sizecode; pc++)
        {
               Instruction i = f->code[pc];
               OpCode o = GET_OPCODE(i);
               SET_OPCODE(i, pnOpTbl[o]);
               f->code[pc] = i;
        }
}
```

修复二

问题2： 在文件头部 反编译出现错误 -- DECOMPILER ERROR: Overwrote pending register.

解决2： 分析发现，原来是解析OP_VARARG错误导致的。OP_VARARG主要的作用是复制B-1个参数到A寄存器中，而反编译工具复制了B个参数，多了一个。修改后的代码如下：

```
...
         case OP_VARARG: // Lua5.1 specific.         {
            int i;
            /*
             * Read ... into register.
             */            if (b==0) {
                TRY(Assign(F, REGISTER(a), "...", a, 0, 1));
            } else {
                 // add by littleNA                // for(i = 0;i<b;i++) {                 for(i = 0; i < b
-1; i++) {
                      TRY(Assign(F, REGISTER(a+i), "...", a+i, 0, 1));
                 }
            }
            break;
         }
...
```

修复三

问题3： 在解析table出现反编译错误 -- DECOMPILER ERROR: Confused about usage of 。registers!

解决3： 分析发现，这里的OP_NEWTABLE 的c参数表示hash table中key的大小，而反编译代码中将c参数进行了错误转换，导致解析错误，修改代码如下：

```
// add by littleNA//#define fb2int(x)    (((x) & 7) << ((x) >> 3))#define fb2int(x)      ((((x) & 7)^8) >> (
((x) >> 3)-1))
```

修复四

问题4： 反编译工具出错并且退出。

解决4： 跟踪发现是在AddToTable函数中，当keyed为0时会调用PrintTable，而PrintTable释放了table，下次再调用table时内存访问失败，修改代码如下：

```
void AddToTable(Function* F, DecTable * tbl, char *value, char *key){
   DecTableItem *item;
   List *type;
   int index;
   if (key == NULL) {
      type = &(tbl->numeric);
      index = tbl->topNumeric;
      tbl->topNumeric++;
   } else {
      type = &(tbl->keyed);
      tbl->used++;
      index = 0;
   }
   item = NewTableItem(value, index, key);
   AddToList(type, (ListItem *) item);
   // FIXME: should work with arrays, too   // add by littleNA   // if(tbl->keyedSize == tbl->used && tbl-
>arraySize == 0){   if (tbl->keyedSize != 0 && tbl->keyedSize == tbl->used && tbl->arraySize == 0) {
      PrintTable(F, tbl->reg, 0);
      if (error)
         return;
   }
}
```

修复五

问题5： 当函数是多值返回结果并且赋值于多个变量时反编译错误，情况如下（lua反汇编）：

```
 21 [-]: GETGLOBAL R0 K9        ; R0 := memoryStatMap 22 [-
]: GETGLOBAL R1 K9        ; R1 := memoryStatMap 23 [-]: GETGLOBAL R2 K2        ; R2 := preload 24 [-
]: GETTABLE  R2 R2 K3     ; R2 := R2["utils"] 25 [-
]: GETTABLE  R2 R2 K16    ; R2 := R2["getCocosStat"] 26 [-]: CALL      R2 1 3       ; R2,R3 := R2() 27 [-
]: SETTABLE  R1 K15 R3    ; R1["cocosTextureBytes"] := R3 28 [-
]: SETTABLE  R0 K14 R2    ; R0["cocosTextureCnt"] := R2
```

当上面的代码解析到27行时，从寄存器去取R3时报错，原因是前面的call返回多值时，只是在F->Rcall中进行了标记，没有在寄存器中标记，编译的结果应该为：

```
memoryStatMap.cocosTextureCnt, memoryStatMap.cocosTextureBytes = preload.utils.getCocosStat()
```

解决5： 当reg为空时并且Rcall不为空，增加一个return more的标记，修改2个函数：

```
char *RegisterOrConstant(Function * F, int r)
{
   if (IS_CONSTANT(r)) {
      return DecompileConstant(F->f, r - 256); // TODO: Lua5.1 specific. Should change to MSR!!!   } else {
      char *copy;
      char *reg = GetR(F, r);
      if (error)
         return NULL;

        // add by littleNA        // if(){}        if (reg == NULL && F->Rcall[r] != 0)
        {
            reg = "return more";
        }

      copy = malloc(strlen(reg) + 1);
      strcpy(copy, reg);
      return copy;
   }
}


void OutputAssignments(Function * F){
   int i, srcs, size;
   StringBuffer *vars;
   StringBuffer *exps;
   if (!SET_IS_EMPTY(F->tpend))
      return;
   vars = StringBuffer_new(NULL);
   exps = StringBuffer_new(NULL);
   size = SET_CTR(F->vpend);
   srcs = 0;
   for (i = 0; i < size; i++) {
      int r = F->vpend->regs[i];
      if (!(r == -1 || PENDING(r))) {
         SET_ERROR(F,"Attempted to generate an assignment, but got confused about usage of registers");
         return;
      }

      if (i > 0)
         StringBuffer_prepend(vars, ", ");
      StringBuffer_prepend(vars, F->vpend->dests[i]);

      if (F->vpend->srcs[i] && (srcs > 0 || (srcs == 0 && strcmp(F->vpend-
>srcs[i], "nil") != 0) || i == size-1)) {
                // add by littleNA                // if()                if (strcmp(F->vpend-
>srcs[i], "return more") != 0)
                {
                        if (srcs > 0)
                               StringBuffer_prepend(exps, ", ");
                        StringBuffer_prepend(exps, F->vpend->srcs[i]);
                        srcs++;
               }
      }

   }
...
}
```

修复六

问题6： 当函数只有一个renturn的时候会反编译错误。

解决6：

```
 case OP_RETURN:
{
    ...
    // add by littleNA    // 新增的if    if (pc != 0)
    {
        for (i = a; i < limit; i++) {
                char* istr;
                if (i > a)
                        StringBuffer_add(str, ", ");
                istr = GetR(F, i);
                TRY(StringBuffer_add(str, istr));
        }
        TRY(AddStatement(F, str));
    }
    break;
}
```

修复七

问题7： 部分table初始化会出错。

解决7：

```
char *GetR(Function * F, int r)
{
   if (IS_TABLE(r)) {
     // add by littleNA        return "{ }";
    //  PrintTable(F, r, 0);    //  if (error) return NULL;   }
...
}
```

修复八

问题8： 可变参数部分解析出错，但是工具反编译时是不报错误的。

解决8： is_vararg为7时，F->freeLocal多加了一次：

```
   if (f->is_vararg==7) {
      TRY(DeclareVariable(F, "arg", F->freeLocal));
      F->freeLocal++;
   }
   // add by littleNA   // 修改if为else if   else if ((f->is_vararg&2) && (functionnum!=0)) {
      F->freeLocal++;
   }
```

修复九

问题9： 反编译工具输出的中文为url类型的字符（类似 “\230\176\148\231\150\151\230\156\175”），不是中文。

解决9： 在proto.c文件中的DecompileString函数中，注释掉default 转换字符串的函数：

```
char *DecompileString(const Proto * f, int n)
{
...
        default:
             //add by littleNA//            if (*s < 32 || *s > 127) {//               char* pos = &
(ret[p]);//               sprintf(pos, "\\%d", *s);//               p += strlen(pos);//            } else {
             ret[p++] = *s;//            }            break;
...
}
```

然后再下面3处增加判断的约束条件，因为中文字符的话，char字节是负数，这样isalpha和isalnum函数就会出错，所以增加约束条件，小于等于127：

```
void MakeIndex(Function * F, StringBuffer * str, char* rstr, int self){
...
   int dot = 0;
   /*
    * see if index can be expressed without quotes
    */   if (rstr[0] == '\"') {

      // add by littleNA      // (unsigned char)(rstr[1]) <= 127 &&      if ((unsigned char)
(rstr[1]) <= 127 && isalpha(rstr[1]) || rstr[1] == '_') {
         char *at = rstr + 1;
         dot = 1;
         while (*at != '"') {

            // add by littleNA            // *(unsigned char*)at <= 127 &&            if (*
(unsigned char*)at <= 127 && !isalnum(*at) && *at != '_') {
               dot = 0;
               break;
            }
            at++;
         }
      }
   }
....
}

...
    case OP_TAILCALL:
    {
               // add by littleNA              // (unsigned char)
(*at) <= 127 &&               while (at > astr && ((unsigned char)
(*at) <= 127 && isalpha(*at) || *at == '_')) {
                  at--;
               }
    }
...
```

修复十

问题10： 反汇编失败。因为一些文件中含有很长的字符串，导致sprintf函数调用失败。

解决10： 增加缓存的大小：

```
void luaU_disassemble(const Proto* fwork, int dflag, int functions, char* name) {
 ...
                        // add by littleNA                        // char lend[MAXCONSTSIZE+128];
          char lend[MAXCONSTSIZE+2048];
...
}
```

修复十一

问题11： op_setlist操作码当b==0时，反编译失败。

解决11： 当遇到类似下面的lua语句时，反编译工具会失败，出现的情况在@lib_ui.lua文件中：

```
local a={func()}
```

汇编后的代码：

```
                a   b   c
[1] newtable    0   0   0    ; array=0, hash=0[2] getglobal  1   0        ; func
[3] call        1   1   0[4] setlist    0   0   1    ; index 1 to top
[5] return      0   1
```

出现的问题有2处，第一个是newtable，当b == 0 && c == 0时，反编译工具认为table是空的table，直接输出了table并且释放了table的内存，导致后面setlist初始化table时找不到内存而报错。

第二个是setlist有问题，当b==0时，其实是指寄存器a+1到栈顶（top）的值全部赋值于table，而反编译器没有对b==0的判断，加上就可以了。所以修改如下：

```
void StartTable(Function * F, int r, int b, int c){
   DecTable *tbl = NewTable(r, F, b, c);
   AddToList(&(F->tables), (ListItem *) tbl);
   F->Rtabl[r] = 1;
   F->Rtabl[r] = 1;
   if (b == 0 && c == 0) {

           // add by littleNA           // for(){}           for (int npc = F->pc + 1; npc < F->f-
>sizecode; npc++)
           {
                  Instruction i = F->f->code[npc];
                  OpCode o = GET_OPCODE(i);
                  if ((o != OP_SETLIST && o != OP_SETTABLE) && r == GETARG_A(i))
                  {
                          PrintTable(F, r, 1);
                          return;
                  }
                  else if ((o == OP_SETLIST || o == OP_SETTABLE) && r == GETARG_A(i))
                  {
                          return;
                  }
           }
      PrintTable(F, r, 1);
      if (error)
         return;
   }
}void SetList(Function * F, int a, int b, int c){
...
   // add by littleNA   // if(){}   if (b == 0)
   {
           Instruction i = F->f->code[F->pc-1];
           OpCode o = GET_OPCODE(i);
           if (o == OP_CALL)
           {
                  int aa = GETARG_A(i);
                  for (i = a + 1; i < aa + 1; i++)
                  {
                          char* rstr = GetR(F, i);
                          if (error)
                                 return;
                          AddToTable(F, tbl, rstr, NULL);
                          if (error)
                                 return;
                  }
           }
           else           {
                  for (i = 1;;i++) {
                          char* rstr = GetR(F, a + i);
                          if (rstr == NULL)
                                 return;
                          AddToTable(F, tbl, rstr, NULL);
                          if (error)
                                 return;
                  }
           }
   }
...
}
```

StartTable 增加的for循环表示，如果执行了newtable(r 0 0)，后面非初始化table的操作覆盖了r寄存器（把table覆盖了），那就表明new出来的table是空的，后面没有对table的赋值；如果后面有对r寄存器初始化，证明此时new出了的table不是空的，是可变参数的table。

SetList 增加的if表示，如果指令是call指令，那么将a+1到call指令寄存器aa的栈元素加入到table中（这里为何不是到栈顶的元素而是到aa的元素呢？因为call指令对应的是函数调用，反编译工具已经把函数调用的字符串解析到aa中了，这里跟实际运行可能有点不一样；else后面就是将a+1到栈顶的元素初始化到table中，直到GetR函数为空表示到栈顶了。

修复十二

问题12： 当一个函数开头只是局部变量声明，如：

```
function func()     local a,b,c
     c = f(a,b)
     return cend
```

第一行 local a,b,c 会反编译失败，导致后面的代码出现各种错误。

解决12：

```
void DeclareLocals(Function * F){
...
   for (i = startparams; i < F->f->sizelocvars; i++) {
      if (F->f->locvars[i].startpc == F->pc) {
             ...
             if (PENDING(r)) {...}
             // add by littleNA             // else if(){}             else if (locals == 0 && F->pc == 0)
             {
                     StringBuffer_add(str, LOCAL(i));
                     char *szR = GetR(F, r);
                     StringBuffer_add(rhs, szR==NULL?"nil":szR);
             }
             ...
       }
   }
...
}
```

当变量的startpc 等于 当前pc，变量的个数为0并且当前pc为0，表示第一行声明了变量，添加的else if就是解析这种情况的（原来是直接报错不解析）。

总结

上文首先总结了近年来公开的lua逆向技术相关文章和相关工具，接着讲解了lua反汇编和反编译的对抗，并以3个实例作为说明。第1个例子举例了征途手游的修复，第2个例子修复了lua虚拟机的opcode并成功反编译lua脚本，第3个例子完美修复了梦幻手游的lua脚本反编译出现的大量错误。

lua加解密的技术还是会一直发展下去，但是这篇文章到此就结束了。接下来可能会写一篇2018腾讯游戏安全竞赛的详细分析报告（详细到每一个字节喔），内容包括但不限于STL逆向、AES算法分析、Blueprint脚本分析等等，敬请期待。

参考文章

[1] 飞虫 《Lua程序逆向之Luac文件格式分析》 https://www.anquanke.com/post/id/87006

[2] 飞虫 《Lua程序逆向之Luac字节码与反汇编》 https://www.anquanke.com/post/id/87262

[3] 飞虫 《Lua程序逆向之Luajit文件格式》 https://www.anquanke.com/post/id/87281

[4] 飞虫 《Lua程序逆向之Luajit字节码与反汇编》 https://www.anquanke.com/post/id/90241

[5] Nick Cano 《Hooking LuaJIT》 https://nickcano.com/hooking-luajit

[6] 興趣使然的小胃 《看我如何通过hook攻击LuaJIT》 https://www.anquanke.com/post/id/86958

[7] Ganlv 《lua脚本解密1：loadstring》 https://www.52pojie.cn/thread-694364-1-1.html

[8] unity 《【放置江湖】LUA手游 基于HOOK 解密修改流程》 https://www.52pojie.cn/thread-682778-1-1.html

[9] 游戏安全实验室 《Lua游戏逆向及破解方法介绍》 http://gslab.qq.com/portal.php?mod=view&aid=173

[10] lNightElf 《[原创]Lua脚本反编译入门之一》 https://bbs.pediy.com/thread-186530.htm

[11] wmsuper 《开心消消乐lua脚本解密》 https://www.52pojie.cn/thread-611248-1-1.html

[12] littleNA 《浅析android手游lua脚本的加密与解密》 https://litna.top/2018/07/07/浅析android手游lua脚本的加密与解密/

[13] 《看雪 2016CrackMe 第二题》 https://ctf.pediy.com/game-fight-3.htm

[14] 《看雪 2017CrackMe 第十五题》 https://ctf.pediy.com/game-fight-45.htm

[15] 《腾讯游戏安全技术竞赛》 https://www.52pojie.cn/forum-77-1.html

[16] luadec https://github.com/viruscamp/luadec

[17] ljd https://github.com/NightNord/ljd

[18] luajit-decomp https://github.com/bobsayshilol/luajit-decomp

[19] 云风 《Lua源码欣赏》 https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/luadec/云风-lua源码欣赏-lua-5.2.pdf

（全文完，谢谢阅读）