

## 0x01 LLVM简介

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

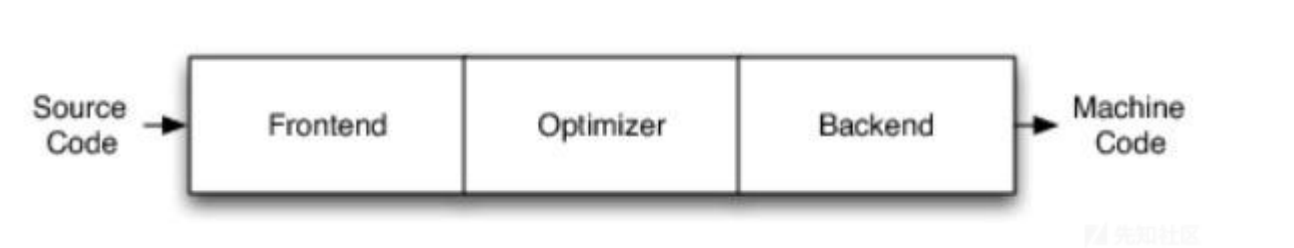
LLVM是模块化、可重用的编译器以及工具链的集合，有些人把LLVM当成是一个低层的虚拟机(low level virtual machine)，但官方给出的解释是这样的：

The name "LLVM" itself is not an acronym; it is the full name of the project.

也就是说LLVM并不是一个缩写，而是整个项目的全名。

LLVM和传统的编译器(GCC)是有差别的

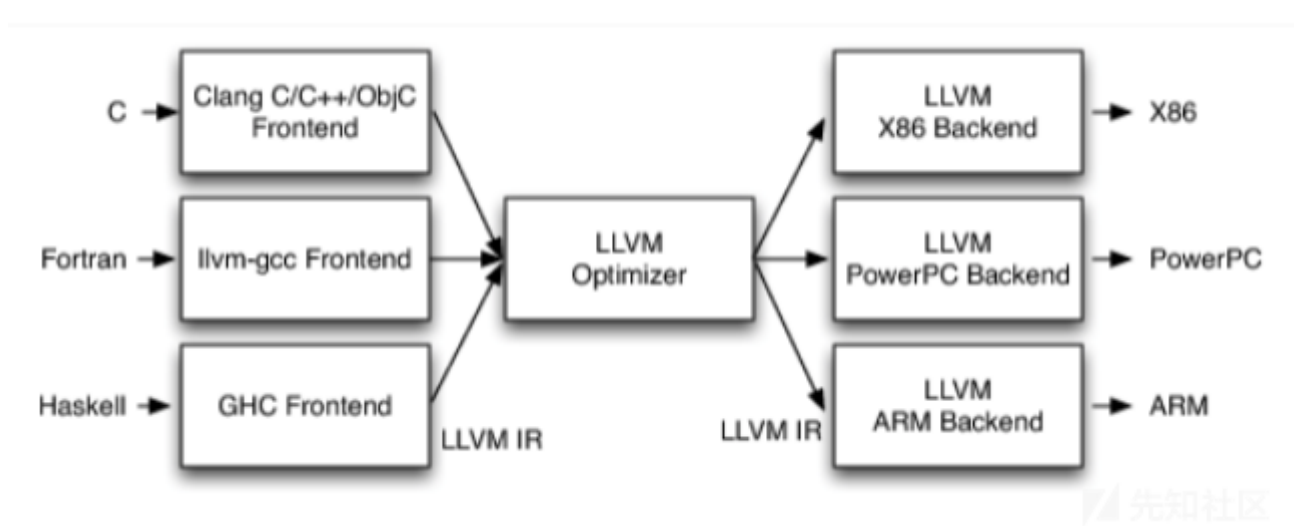
传统的编译器架构



传统的编译器架构主要分为三个部分

- Frontend:前端  
包括词法分析、语法分析、语义分析、中间代码生成
- Optimizer:优化器  
主要是对编译前端对生成的中间代码的优化
- Backend:后端  
翻译中间代码为native机器码

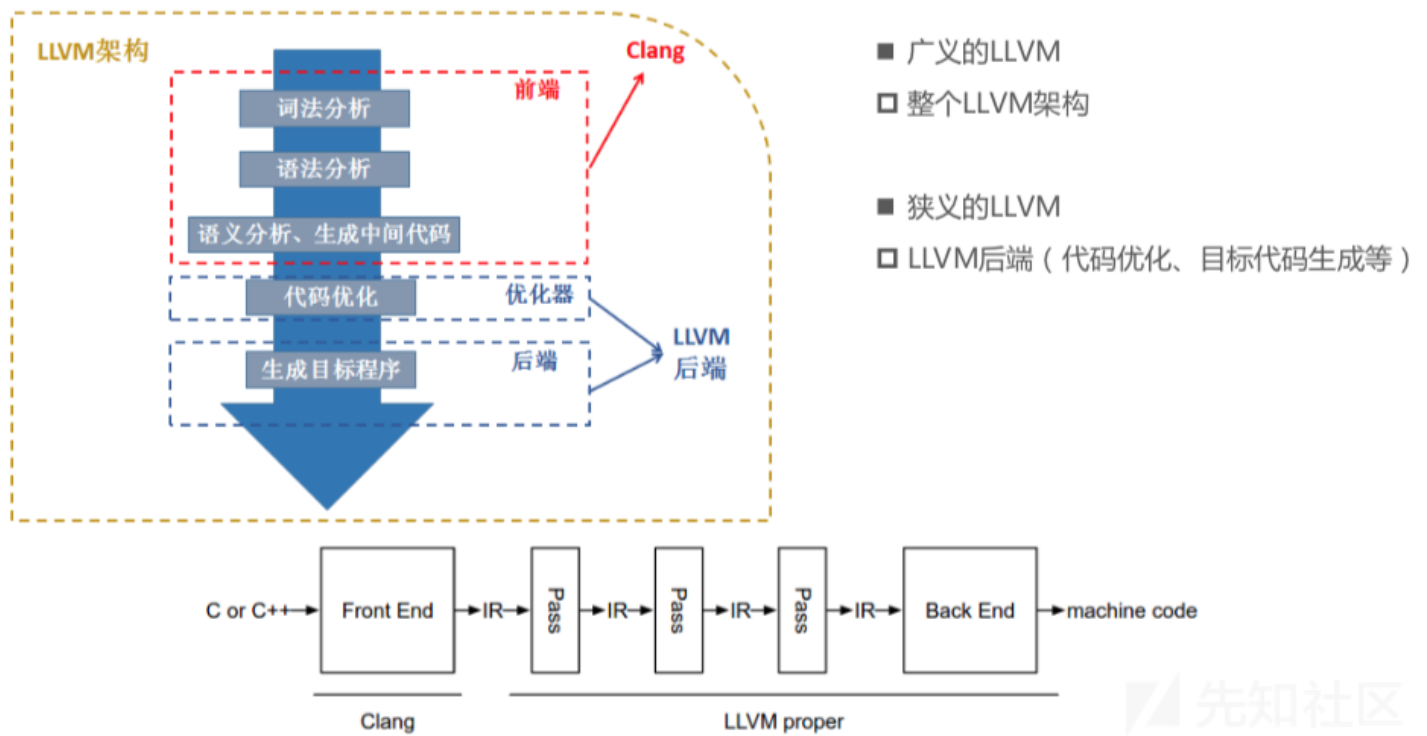
LLVM编译器架构



LLVM编译器套件与传统编译器架构的不同之处主要在于

- LLVM编译器的前端其它层(优化器、后端)是分离的，LLVM专门有一个Clang子项目用来对源码进行编译，生成IR(intermediate representation)中间字节码;而传统编译器的代表(GCC)由于编译前后端耦合度太高，增加一个前端语言支持或者一个后端平台支持将会变得异常复杂。相比之下LLVM由
- LLVM编译器不同的前端统一使用相同的中间码，不像GCC有各种风格(intel&ATT)
- LLVM经常被用于一些解释型语言的动态编译(优化)。类似的JAVA虚拟机(JVM)的JIT(好像现在就有厂在做基于LLVM的JAVA JIT编译器，负责将高层字节码(java-bytecode)解析成相对底层的IR中间码，之后编译成相应平台的机器码执行。
- LLVM也经常被用于一些语言的静态编译，类似的Objective-c就是使用Clang进行编译(之前其实也是使用GCC的,但现在连Xcode的内置编译器都换成Clang了)，据说编译

## 0x02 狭义的LLVM和广义的LLVM



广义的LLVM通常指LLVM编译器的整体架构，而狭义的LLVM通常指不包含前端，只实现中间代码优化和native码生成的部分。IR中间码需要多个pass进行一系列优化后再进

### 0x03 字节码抽象层次

比较典型的就是java bytecode与LLVM IR之间的抽象层次比较，java bytecode与LLVM IR都是用于描述代码运算的模型，但两者的抽象层次是不同的。之前想过一个问题，就是为什么编译器/虚拟机需要引入中间码/字节码，现在大概可以理解，源码通过编译前

### 0x04 OLLVM

LLVM前端是Clang，当对源代码进行编译生成IR中间码以后，优化器会对IR进行优化，然后后端生成执行代码。试想一下如果IR进行优化的过程可控，那么LLVM编译后端生下面是一些用于混淆的成熟开源项目，打算之后来一波源码分析。

[OLLVM](#)  
[Hikari](#)  
[Armariris\(孤挺花\)](#)

#### 1、编译

# Installation

R1kk3r edited this page on 29 Jun 2017 · 19 revisions

You will find here some informations on how to get, build and use our project.

## 🔗 Getting the sources and building

We maintain several branches: the one named `llvm-4.0` is the latest official (i.e., non-dev) version and is based on the latest version 4.0.1 released by the LLVM team. Older branches are also available: `llvm-3.3`, `llvm-3.4`, `llvm-3.5`, `llvm-3.6.1`

At the moment, all our obfuscation transforms have been ported in all branches.

To get the latest version of the LLVM branch, you can use the following commands:

```
$ git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator.git
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ../obfuscator/
$ make -j7
```

Older branches can be accordingly be cloned.

When the build is finished, you should have all the binaries in `build/bin`. Note that this source contain LLVM and Clang.

这边有个坑，编译的时候说xxx已经存在，看dalao博客找到的编译选项，可以正常编译

```
cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_INCLUDE_TESTS=OFF ../obfuscator/
```

然后就是缓慢的编译过程。。。。。

```
Ubuntu_16.04_Fuzzer x
make-j7 xdw
[ 28%] Building CXX object tools/llvm-profdata/CMakeFiles/llvm-profdata.dir/llvm-profdata.cpp.o
[ 28%] Building CXX object projects/compiler-rt/lib/asan/CMakeFiles/RTAsan.i386.dir/asan_wtn.cc.o
[ 28%] Built target RTAsan.i386
Scanning dependencies of target lli-child-target
[ 28%] Building CXX object tools/lli/ChildTarget/CMakeFiles/lli-child-target.dir/ChildTarget.cpp.o
[ 28%] Building CXX object utils/TableGen/CMakeFiles/llvm-tblgen.dir/CodeGenInstruction.cpp.o
Scanning dependencies of target llvm-mcmarkup
[ 28%] Building CXX object tools/llvm-mcmarkup/CMakeFiles/llvm-mcmarkup.dir/llvm-mcmarkup.cpp.o
[ 28%] Building CXX object tools/lli/ChildTarget/CMakeFiles/lli-child-target.dir/_/_/RemoteTarget.cpp.o
[ 28%] Linking CXX executable ../../bin/llvm-config
[ 28%] Linking CXX executable ../../bin/llvm-profdata
[ 28%] Built target llvm-config
Scanning dependencies of target clang-tblgen
[ 28%] Built target llvm-profdata
[ 28%] Building CXX object utils/TableGen/CMakeFiles/llvm-tblgen.dir/CodeGenMapTable.cpp.o
[ 28%] Building CXX object tools/clang/utils/TableGen/CMakeFiles/clang-tblgen.dir/ClangASTNodesEmitter.cpp.o
[ 28%] Linking CXX executable ../../bin/lli-child-target
[ 28%] Built target lli-child-target
Scanning dependencies of target clang_rt.asan-i386
[ 28%] Linking CXX static library ../../bin/libclang_rt.asan-i386.a
[ 28%] Built target clang_rt.asan-i386
Scanning dependencies of target tsan
[ 28%] Built target tsan
Scanning dependencies of target msan
[ 28%] Built target msan
Scanning dependencies of target RTAsanTest.i386-inline
[ 28%] Linking CXX static library libRTAsanTest.i386-inline.a
[ 28%] Linking CXX executable ../../bin/llvm-mcmarkup
[ 28%] Built target RTAsanTest.i386-inline
Scanning dependencies of target RTAsanTest.i386-with-calls
[ 28%] Linking CXX static library libRTAsanTest.i386-with-calls.a
[ 28%] Building CXX object utils/TableGen/CMakeFiles/llvm-tblgen.dir/CodeGenRegisters.cpp.o
[ 28%] Built target llvm-mcmarkup
[ 28%] Built target RTAsanTest.i386-with-calls
[ 28%] Building CXX object tools/clang/utils/TableGen/CMakeFiles/clang-tblgen.dir/ClangAttrEmitter.cpp.o
Scanning dependencies of target asan
[ 28%] Built target asan
[ 28%] Building CXX object tools/clang/utils/TableGen/CMakeFiles/clang-tblgen.dir/ClangCommentCommandInfoEmitter.cpp.o
[ 28%] Building CXX object utils/TableGen/CMakeFiles/llvm-tblgen.dir/CodeGenSchedule.cpp.o
[ 28%] Building CXX object tools/clang/utils/TableGen/CMakeFiles/clang-tblgen.dir/ClangCommentHTMLNamedCharacterReferenceEmitter.cpp.o
[ 28%] Building CXX object tools/clang/utils/TableGen/CMakeFiles/clang-tblgen.dir/ClangCommentHTMLTagsEmitter.cpp.o
[ 28%] Building CXX object utils/TableGen/CMakeFiles/llvm-tblgen.dir/CodeGenTarget.cpp.o
[ 28%] Building CXX object tools/clang/utils/TableGen/CMakeFiles/clang-tblgen.dir/ClangDiagnosticsEmitter.cpp.o
[ 28%] Building CXX object tools/clang/utils/TableGen/CMakeFiles/clang-tblgen.dir/ClangSACheckersEmitter.cpp.o
[ 29%] Building CXX object tools/clang/utils/TableGen/CMakeFiles/clang-tblgen.dir/NeonEmitter.cpp.o
[ 29%] Building CXX object tools/clang/utils/TableGen/CMakeFiles/clang-tblgen.dir/TableGen.cpp.o
Scanning dependencies of target compiler-rt
[ 29%] Built target compiler-rt
[ 29%] Building CXX object utils/TableGen/CMakeFiles/llvm-tblgen.dir/DAGISelEmitter.cpp.o
[ 29%] Building CXX object utils/TableGen/CMakeFiles/llvm-tblgen.dir/DAGISelMatcherEmitter.cpp.o
[ 29%] Building CXX object utils/TableGen/CMakeFiles/llvm-tblgen.dir/DAGISelMatcherGen.cpp.o
[ 29%] Building CXX object utils/TableGen/CMakeFiles/llvm-tblgen.dir/DAGISelMatcherOpt.cpp.o
```

要将输入定向到该虚拟机，请将鼠标指针移入其中或按 Ctrl+G。

orz.....焦灼

编译到百分之三十多的时候，突然进度条就卡住了，然后编译过程崩溃，查了一下好像是内存分配的太少了，导致进程卡死，于是给虚拟机加了两个G，继续编译，发现报错

g++: internal compiler error: Killed (program cc1plus)

查了下资料发现还是内存不足。。。 (我都给了四个G了) 解决方法是加一个临时的交换分区

```
sudo dd if=/dev/zero of=/swapfile bs=64M count=16
sudo mkswap /swapfile
sudo swapon /swapfile
After compiling, you may wish toCode:
sudo swapoff /swapfile
sudo rm /swapfile
```

最后不容易终于编译成功了，build/bin目录下生成了编译前端

```
Ubuntu_16.04_Fuzzer x
xdw@ubuntu: ~/build/bin
→ ~ ls
Octf_2019 build Desktop Downloads Exploit how2heap libc-database obfuscator peda Pictures PROJ pwnable.tw pwn_xman_2018 test utils Videos
bin build2 Documents examples.desktop Fuzzer ida_server Music obfuscator_4 pedty Pin Public pwndbg Templates test.c Valgrind_tools VM_Easy
→ ~ cd build
→ build ls
bin CMakeCache.txt cmake_install.cmake CPackConfig.cmake docs examples lib LLVMBuild.cmake projects share utils
cmake CMakeFiles compile_commands.json CPackSourceConfig.cmake DummyConfigureOutput include libexec Makefile runtimes tools
→ build cd bin
→ bin ls
arc4e-test clang-4.0 clang-offload-bundler lli llvm-config llvm-dis llvm-link llvm-nodextract llvm-profdata llvm-stress obj2yaml verify-uselistorder
bugpoint clang-check clang-tblgen lli-child-target llvm-cov llvm-dsymutil llvm-lit llvm-nm llvm-ranlib llvm-strings opt yaml2obj
c-arcot-test clang-cl count llvm-ar llvm-c-test llvm-dwarfdump llvm-lto llvm-obdump llvm-readobj llvm-symbolizer sancov yaml-bench
c-index-test clang-cpp diagtool llvm-as llvm-cxxdump llvm-dwp llvm-lto2 llvm-opt-report llvm-rtdyld llvm-tblgen sanstats
clang clang-format filecheck llvm-bcanalyzer llvm-cxxfilt llvm-extract llvm-mc llvm-pdbdump llvm-size llvm-xray scan-build
clang++ clang-import-test tlc llvm-cat llvm-diff llvm-lib llvm-mcmarkup llvm-PerfectShuffle llvm-split not scan-view
```

## 2、混淆参数

OLLVM有一些混淆参数，类似的有字符串加密、控制流扁平化、指令替换、控制流伪造等等

### 1、控制流扁平化

```
clang -mllvm -fla test.c -o test1
```

## 2、指令替换

```
clang -mllvm -sub test.c -o test2
```

## 3、控制流伪造

```
clang -mllvm -bcf test.c -o test3
```


对比一下混淆编译之后的可执行文件大小

```
→ ~ cd test0bfuse
→ test0bfuse ls
test test1 test2 test3 test.c
→ test0bfuse ls -alh
total 60K
drwxrwxr-x  2 xdw xdw 4.0K Apr  2 20:26 .
drwxr-xr-x 44 xdw xdw 4.0K Apr  2 20:26 ..
-rwxrwxr-x  1 xdw xdw 8.6K Apr  2 05:55 test
-rwxrwxr-x  1 xdw xdw 8.5K Apr  2 20:24 test1
-rwxrwxr-x  1 xdw xdw 8.5K Apr  2 20:25 test2
-rwxrwxr-x  1 xdw xdw 8.6K Apr  2 20:25 test3
-rw-rw-r--  1 xdw xdw  189 Apr  2 05:55 test.c
→ test0bfuse ls -alh
total 60K
drwxrwxr-x  2 xdw xdw 4.0K Apr  2 20:26 .
drwxr-xr-x 44 xdw xdw 4.0K Apr  2 20:26 ..
-rwxrwxr-x  1 xdw xdw 8.6K Apr  2 05:55 test
-rwxrwxr-x  1 xdw xdw 8.5K Apr  2 20:24 test1
-rwxrwxr-x  1 xdw xdw 8.5K Apr  2 20:25 test2
-rwxrwxr-x  1 xdw xdw 8.6K Apr  2 20:25 test3
-rw-rw-r--  1 xdw xdw  189 Apr  2 05:55 test.c
→ test0bfuse █
```

在文件比较小的情况下好像差别并不明显2333333，OLLVM牛逼(滑稽)。

## 3、混淆效果

先贴一下test.c的源码

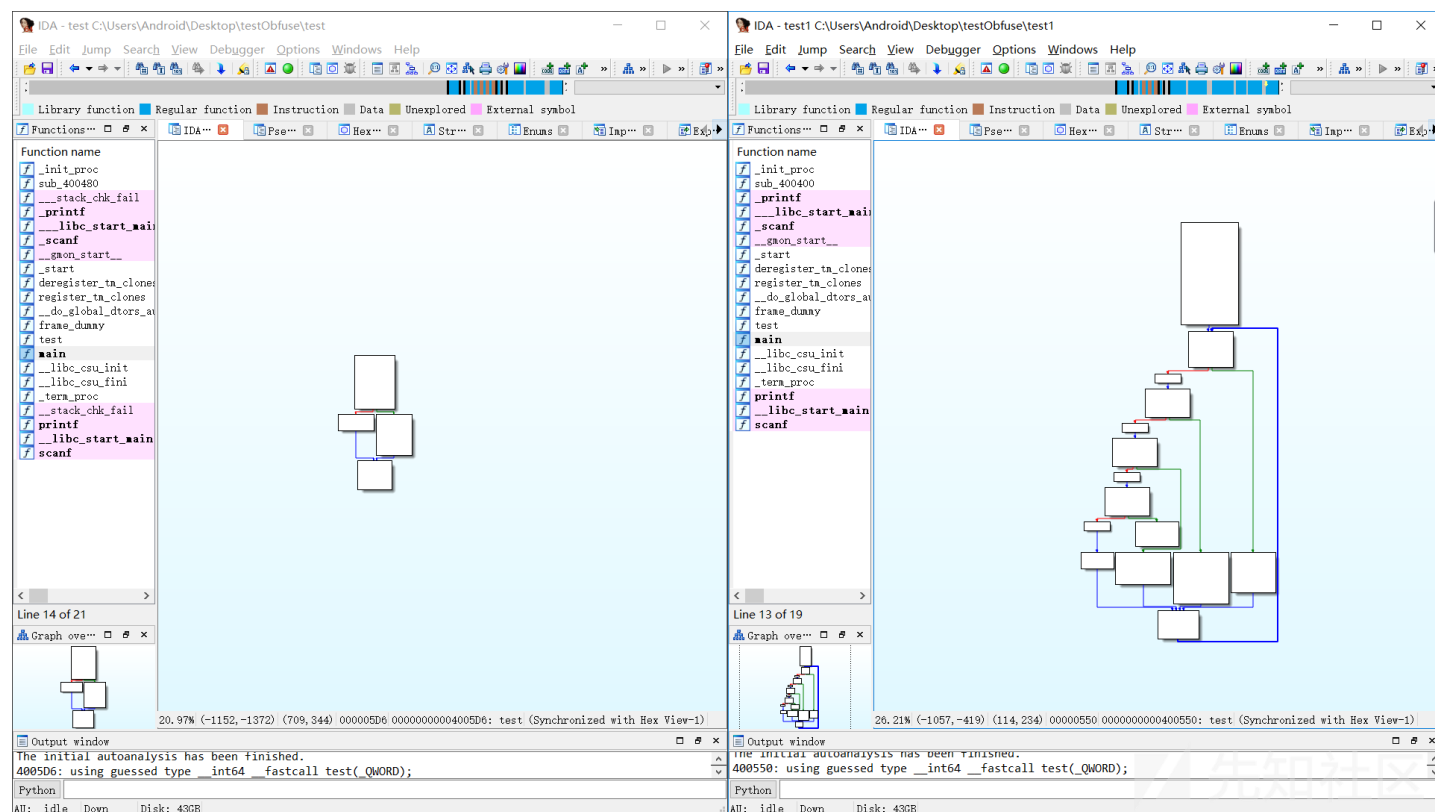
 C:\Users\Android\Desktop\testObfuse\test.c - Notepad++

```
文件(F) 编辑(E) 搜索(S) 视图(V) 编码(N) 语言(L) 设置(T) 工具(O) 宏(M) 运行(R) 插件(P) 窗口(W) ?

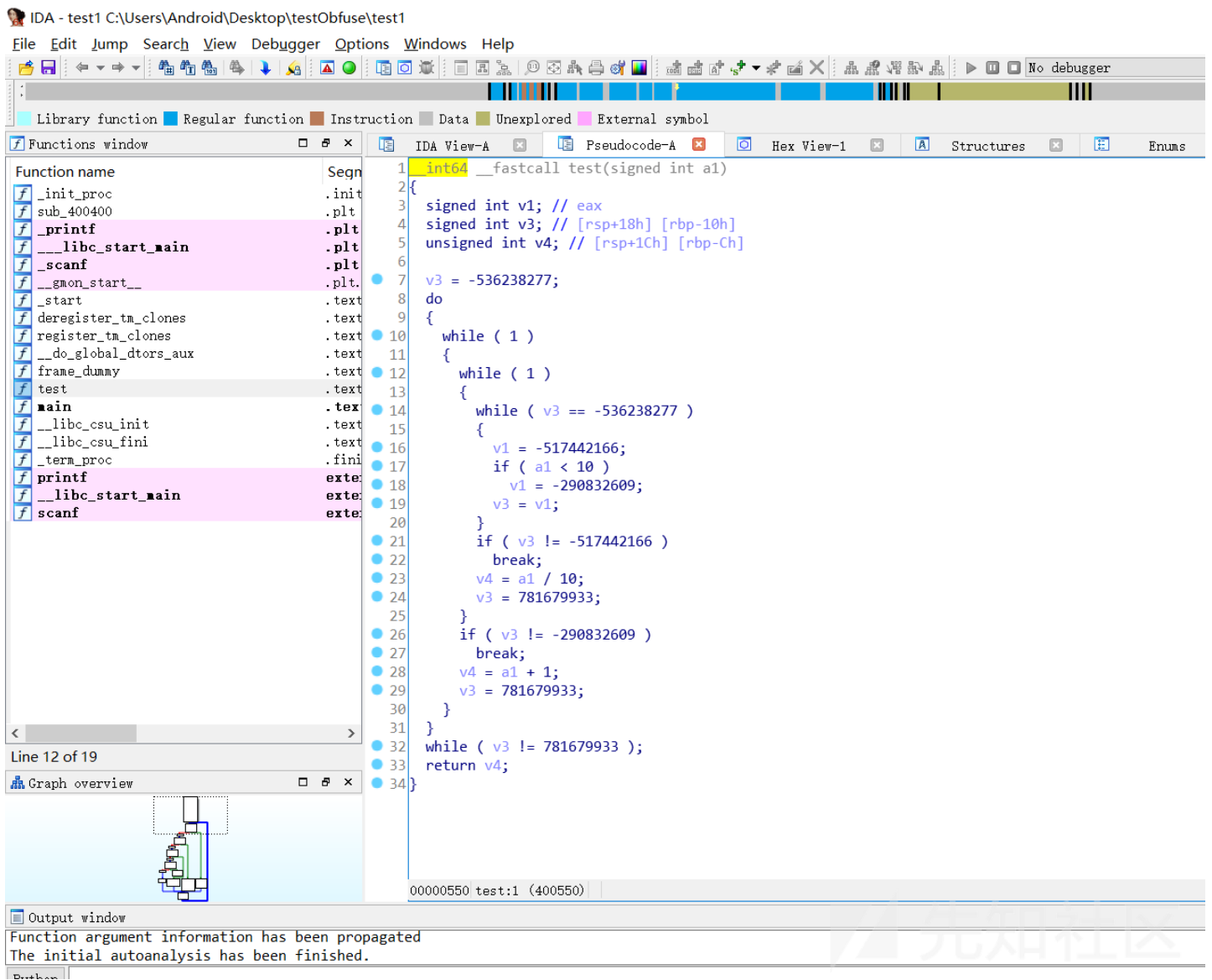
config.json x README.md x 面试.txt x 米哈游一面.MD x 二面准备.md x 其它常识.MD x

1 int test(int a1){
2     int result;
3     if(a1<10){
4         result=a1+1;
5     }
6     else{
7         result=a1/10;
8     }
9     return result;
10 }
11
12 int main(){
13     int r,s;
14     scanf("%d",&s);
15     r=test(s);
16     printf("%d\n",r);
17     return 0;
18 }
19
```

在ida里面看一下混淆以后的效果



右边是开启控制流扁平化以后的程序的ida视图，左边是未添加编译保护的程序的控制流图



可以看到程序逻辑至少复杂了一个量级，而且一些常量被替换了，导致分析起来也觉得难以理解。OLLVM牛逼！！

## 0x05 总结

这篇文章只是记录了一下学习LLVM&OLLVM的过程，其实说实话并没有进行比较详细的分析，还有一些拓展面也没有想好怎么写，等下次再填坑吧。

点击收藏 | 1 关注 | 1

[上一篇：BugBounty：防火墙与缓存机...](#) [下一篇：libpng历史漏洞分析](#)

1. 2 条回复



[公子扶苏呀呀呀](#) 2019-11-05 21:11:40

牛逼牛逼

0 回复Ta

---



[drive\\*\\*\\*\\*](#) 2019-11-05 21:47:53

[@公子扶苏呀呀呀](#) 你更牛逼 欸嘿嘿 啥时候dalao也来一篇 我给dalao点赞

0 回复Ta

---

[登录](#) 后跟帖

先知社区

---

[现在登录](#)

热门节点

---

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)