涉密论文 □　公开论文 □

浙江大学

本 科 生 毕 业 论 文（设计）

题目 基于 AFL 的静态插桩功能扩展的实现

姓名与学号 许晟杰 3140103543

指导教师 陈焰

年级与专业 大四 计算机科学与技术

所在学院 计算机学院

提交日期 2018 年 5 月 30 日

浙江大学本科生毕业论文（设计）承诺书

1. 本人郑重地承诺所呈交的毕业论文（设计），是在指导教师的指导下严格按照学校和学院有关规定完成的。

2. 本人在毕业论文（设计）中除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 浙江大学 或其他教育机构的学位或证书而使用过的材料。

3. 与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

4. 本人承诺在毕业论文（设计）工作过程中没有伪造数据等行为。

5. 若在本毕业论文(设计)中有侵犯任何方面知识产权的行为，由本人承担相应的法律责任。

6. 本人完全了解 浙江大学 有权保留并向有关部门或机构送交本论文（设计）的复印件和磁盘，允许本论文（设计）被查阅和借阅。本人授权 浙江大学 可以将本论文（设计）的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编本论文（设计）。

作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

致　谢

感谢朱梦凡学长、陈焰老师以及上海科技大学的陈浩老师，在我进行我的毕设项目时，对我的启发、指导和帮助。

感谢 Lcamtuf 开发如此优秀的 AFL，带我走进 Fuzzing 的世界。

感谢参考文献中的作者们，他们的文章/代码，让我对本项目有了很好的出发点和参考点。

最后，谢谢论文评阅老师们的辛苦工作。衷心感谢我的家人、朋友，以及同学们，在他们的鼓励和支持下我才得以顺利完成此论文。

摘　要

Fuzzing（模糊测试）作为一种保证程序质量，挖掘现有漏洞的重要技术，在近几年飞速发展。特别是随着程序规模越来越大，程序模糊测试技术得到了越来越广泛的研究和应用，卓有成效。而 ARM 作为全球最被广泛使用的指令集架构[1]，尤其是被使用在各类实时性要求高的生产环境以及各种智能家庭设备中，却由于绝大多数在使用背景下的 ARM 程序没有源代码，仅仅只有编译后的二进制文件，如路由器固件等，分析程序并插桩较为困难，所以模糊测试在 ARM 的应用上却没有太大的进展。在 Fuzzing 技术逐渐成熟的过程中，American Fuzzy lop（AFL）就是一个代表之作，AFL 通过在软件编译时在程序中插入插桩，进行以代码覆盖率为指导的灰盒 Fuzzing，在开源软件的测试中发挥了极其重要的作用。但 AFL 对于无源码情况的程序，仅仅只支持 x86 架构下 Linux 平台 ELF 程序的插桩，而对于 ARM 架构程序则束手无策。在这样的背景下，本项目旨在针对 ARM 架构进行 AFL 的改进或扩展，使其能针对 ARM 架构的无源码程序进行 Fuzzing。通过 IDA 提供的 API，找到程序里的基本块，再利用静态插桩框架，对每一个基本块都进行插桩，插入和 AFL 原先逻辑对接的代码，来达到和 AFL 对开源程序插桩近似的效果。最终经过一定的测试验证，扩展功能能够正常与 AFL 原先逻辑结合，对 ARM 架构的闭源程序进行 Fuzzing 工作。

关键词：模糊测试；ARM 架构；AFL；静态插桩；

Abstract

Fuzzing as an important technology to ensure the quality of code and dig existing bugs has developed rapidly in recent years. In particular, with the increasing scale of the program, the program fuzzing test technology has been more and more extensive research and application, which has achieved remarkable results. ARM as the most widely used instruction set architecture in terms of quantity produced, especially used in various real-time demanding production environments and various smart home devices. But due to the vast majority of applications in the context of the ARM program without source code, only compiled binary files, such as router firmware, etc., it is difficult to analyze the program and implement the instrument. So, fuzz testing has not made much progress in the application of ARM programs. In the process of gradual maturity of Fuzzing technology, the American Fuzzy lop (AFL) is a representative work. AFL inserts instrumentation into the program during software compilation and carries out gray box Fuzzing guided by the code coverage rate in open source software. The test played an extremely important role. The AFL program only supports the Linux platform ELF program for the x86 architecture, but it does nothing for the ARM architecture program. In this context, this project aims to improve or extend the AFL for the ARM architecture, enabling it to perform Fuzzing against ARM-based, non-sourced programs. Through the API provided by IDA, find the basic blocks in the program, and then use the static instrumentation framework to perform instrumentation for each basic block. Inserting some code to communicate with the original logical of the AFL, to achieve similar result compared with AFL open source program instrumentation. After a certain amount of test verification, the extended function can be combined with the original AFL logic to perform Fuzzing work on the closed source program of the ARM architecture.

Key words: Fuzzing、ARM architecture、AFL、Static instrumentation

目 录

第一部分 毕业设计

1 项目背景..... 1
1.1 项目背景及意义..... 1
1.1.1 背景介绍..... 1
1.1.2 本项目的意义..... 2
1.2 项目内容..... 3
1.2.1 静态插桩框架..... 3
1.2.2 基本块的搜索..... 3
1.2.3 与 AFL 对接逻辑..... 4
1.3 本文结构..... 4
2 AFL 静态插桩扩展功能架构概述..... 5
2.1 整体需求概述..... 5
2.2 整体系统结构..... 6
2.2.1 静态插桩框架结构..... 6
2.2.2 与 AFL 逻辑相连接的逻辑架构..... 8
2.2.3 搜索程序基本块..... 9
2.2.4 项目整体逻辑架构..... 10
2.3 关键技术分析..... 11
2.3.1 静态插桩..... 11
2.3.2 基本块的搜索..... 11
2.3.3 AFL Fuzzing..... 12
2.4 本章小结..... 12
3 项目主要工作内容..... 13
3.1 AFL 插桩代码..... 13
3.1.1 需求分析..... 13
3.1.2 架构设计..... 14

3.1.3 实现细节.....15
3.2 对二进制文件进行插桩.....19
3.2.1 需求分析.....19
3.2.2 架构设计.....19
3.2.3 实现细节.....20
3.3 本章小结.....23
4 测试及结果.....24
4.1 对于搜索基本块程序的测试.....24
4.2 对于静态插桩框架的测试.....24
4.3 结合 AFL 的整体测试.....26
4.4 本章小结.....28
5 总结与展望.....28
参考文献.....29
附 录.....30
作者简历.....31

《浙江大学本科生毕业论文（设计）任务书》
《浙江大学本科生毕业论文（设计）考核表》

第二部分 文献综述和开题报告

文献综述和开题报告封面
指导教师对文献综述和开题报告具体内容要求
目录.....I
一、文献综述（中期报告）.....1
二、开题报告.....3
三、外文翻译.....5
四、外文原文
《浙江大学本科生文献综述和开题报告考核表》

第一部分

毕业论文（设计）

1 项目背景

1.1 项目背景及意义

模糊测试（Fuzzing）是传统PC平台上近些年非常流行的软件测试方法，相比较于传统的人工漏洞挖掘，Fuzzing可以提供更高的自动化，在软件测试过程中，持续性的对软件进行测试，并且Fuzzing可以利用随机或者半随机的数据，以覆盖率等作为回馈指导，从而达到挖掘出深层次软件漏洞的目的。

在Fuzzing技术逐渐成熟的过程中，AFL（American Fuzzy lop）就是一个代表之作。AFL通过在软件编译时，在程序中插入插桩代码，以统计程序执行的分支覆盖率，并根据覆盖率的反馈，来决定输入的数据，进而变异输入数据，使得能够探索程序中的更多的分支，如此就有更大的概率可以触发深层次的漏洞。由于涉及代码插桩，并对数据变异进行一定的反馈指导，因而AFL是一种典型的灰盒fuzzer。在实际使用过程中，AFL在开源软件的测试中发挥了极其重要的作用，其编译产生的代码运行效率很高，加上采用Fork-Server等技术提速使得AFL测试的速度可以达到接近原生执行的速度。

但是在很多情况下，并不能直接获取到软件源码。而AFL对于非开源软件，仅仅在x86架构的Linux平台下有相关的支持(QEMU模式)，但由于其利用了QEMU的代码翻译功能和分块功能，也使得 Fuzzing 效率大大降低。

在这样的背景下，本项目扩展了AFL针对ARM架构上的静态插桩以及Fuzzing功能，使其能正常与AFL原先逻辑结合，对ARM架构的闭源程序进行Fuzzing工作。

1.1.1 背景介绍

一些模糊测试工具使用符号执行来解决路径约束[6][7]，但符号执行速度较慢，无法有效地解决许多类型的约束[8]。为了避免这些问题，AFL不使用符号执行或任何重量级程序分析[2]。 它通过程序的反馈来观察哪些输入能触及新的程序分支，并将这些输入保留为进一步突变的种子，然后在这些种子的基础上对输入进行不断变异。

为了提升性能，AFL 使用了Fork-Server技术，使得 Fuzzzing 进程只进行一次execve()，linking 和 libc initialization，之后的 Fuzzzing 进程通

过 fork() 来进行拷贝已经加载好的整个程序，避免了重复初始化程序造成的浪费。而主 Fuzzzing 进程则通过预先定义好的文件描述符来和后fork出的进程进行通讯，传输控制指令和覆盖率数据。

而对于没有源代码的情况，AFL 基于QEMU，开发出了AFL-QEMU 模式，来完成无源码情况下的 Fuzzing。QEMU是一套开源模拟器，可以用来模拟运行多种架构的程序，如x86、MIPS、PowerPC等，在GNU/Linux平台上使用广泛。其工作原理在于其动态代码翻译机制，QEMU使用Basic Blocks作为翻译单元，并在翻译之后把翻译后的代码块放入缓存中，之后便只运行缓存中的代码块而不再重新翻译[9]。而 AFL 对 QEMU 的源代码打了补丁，使得其在每一次翻译新Basic Blocks时，插入AFL自己的代码来完成AFL所需的Fork-Server 以及记录路径并跳转等一系列的工作。这是非常有创造性的一种想法，使得AFL主体并不需要修改太多，就能同时满足有源码和无源码模式下的 Fuzzing。但这样也是有代价的，在速度上，这比有源码模式下要慢2-5倍，但这已经比用DynamoRIO或PIN来完成这样的工作要快得多了[4]。

1.1.2 本项目的意义

本项目着眼于AFL Fuzzing技术的不足，来解决主要以下问题：

1. 开发了针对ARM架构ELF程序的静态插桩框架，能够识别ARM架构ELF程序的各个段和区，并且支持对ARM结构ELF程序进行修改二进制代码、添加段、修改段属性、添加外部库以及导入函数等基本操作。该框架较为通用，不仅仅可以用于AFL静态插桩，一些针对二进制程序的调试、研究等，均可以使用。在本项目中，主要利用该框架来进行插入与AFL沟通的代码的工作。

2. AFL 不支持除了x86 Linux 以外平台的无源码程序 Fuzzing，本项目针对ARM平台的二进制程序，实现类似AFL对开源代码编译执行效果的 Fuzzing，从而使得AFL在ARM平台下能够进行 Fuzzing 漏洞的工作。由于至今，并没有实用性强的针对ARM程序的Fuzzing解决方案，这对于现今在大量生产环境以及智能家居设备中大量使用的ARM架构程序来说，无疑能更好的保证其安全性。

1.2 项目内容

该项目的开发部分分为3个主要部分。

1.2.1 静态插桩框架

与AFL针对开源程序的插桩相比，静态插桩框架直接面向的就是以及编译好的二进制程序。在Github上已经现有许多静态插桩框架，如LIEF等，但要么不支持ARM架构，要么功能极少，只支持替换指定文件偏移的一段代码，面对更长的代码或者是需要外部导入函数，就束手无策。所以本项目先从头开始，开发了一套支持ARM架构ELF程序文件的静态插桩框架。该框架由Python编写，基于Capstone 和Keystone 汇编反汇编框架。Capstone 和Keystone 是两款轻量级的，支持多种CPU架构，支持多个平台，提供汇编反汇编指令的优秀框架,并且有着完善的C/Python API。

本项目开发的静态插桩框架首先解析二进制程序文件格式，识别出程序文件各项属性以及各个段和区,并提供修改二进制代码、添加段、修改段属性、添加外部库以及导入函数等基本操作。

1.2.2 基本块的搜索

AFL插桩时，是在程序的每一个Basic blocks（基本块）的最开始，加入自己的代码，来实现记录程序的执行路径，计算覆盖率，以及跳转。基本块指的是一段代码序列，除了入口和出口以外没有分支[10]，同时基本块也形成了控制流图（CFG）中的顶点和节点，只要执行基本块中的第一条指令，那么基本块中其余指令就必须按顺序执行一次。所以在每个基本块的最开始进行插桩，用来记录Fuzzing的程序运行覆盖率的多少，十分合适。

本项目采用了IDA API来完成基本块搜索这一工作，IDA API是hey-rays开发用来让用户能够自定义扩展IDA原有功能用的，以IDA Python[11]为接口进行编程，能够利用Python完成IDA的一些基础功能。本项目利用其完成了基本块搜索。

除此之外，由于ARM程序内部可能存在2种指令集（ARM指令集和Thumb指令集），这两种指令集区别极大，在静态插桩时需要区别对待，但在每一个基本块里，指令集不会混用。所以本项目在搜索基本块时，也识别出了所有基本块的属性，标记对应基本块使用的是ARM指令集还是Thumb指令集。

1.2.3 与 AFL 对接逻辑

本项目作为AFL的扩展开发，需要与AFL原先逻辑结合，进行ARM程序的Fuzzing。而与AFL的原始逻辑的对接，主要包括setup、fork-server、log 3个部分，分别负责与AFL沟通管道的构建和一些全局数据区的初始化、在程序完全启动后（所有动态库都已经加载）进行多次fork()操作来让子进程作为新的Fuzzing程序、在程序执行到每一个基本块之后记录当前分支信息进行反馈。本项目对所有基本块的最开始插入了汇编代码，来完成这3个部分的操作。

1.3 本文结构

本文第一章主要介绍了本项目的背景和意义，以及本项目的基本内容。

本文第二章则对本项目的架构进行了梳理，在大致阐述了基本架构之后，对各个部分都进行了图文并茂的阐述，再对整体架构进行了简单的描述，来让读者有较为全面的了解，并且也解释了一些开发时候使用的框架和技术，以及使用的原因。

本文第三章将对本项目的具体需求进行细化分析，以及对本项目主要部分的具体实现细节进行展示和解释。主要包括了插入的AFL插桩代码的细节实现，和插桩代码的细节实现。

本文第四章包括对本项目的成果的测试，以及对测试结果的分析。

本文第五章则是对本项目的工作进行了总结和对未来可以进一步改进的工作做了展望。

2 AFL 静态插桩扩展功能架构概述

2.1 整体需求概述

在本项目中，需要实现上文中提到的3个部分的内容。

对于静态插桩框架，需要能够支持各种类型的基本补丁功能，主要包括：修改指定位置（文件偏移/虚拟地址偏移）的汇编指令，在指定位置插入汇编指令，修改段属性，添加新段（至少包括添加新数据段和新代码段），添加外部库以及导入外部库函数。在插入新的汇编代码时，不能影响原先的程序代码在文件中的偏移，这会导致一系列相对寻址问题，如相对跳转和重定向。除此之外，还需要保证在插入汇编指令的前后，上下文保持不变，这一点非常重要，否则在执行插桩代码之后，上下文寄存器如果发生了改变，程序原始的逻辑就会收到影响，程序的流程和走向可能会不再受控制，发生异常的结果。而且，要做到插桩的代码是上下文不敏感，这样在执行插桩代码时，能保证执行代码按照理想的状态运行，而不会被程序原先的逻辑影响。总而言之，要保证静态插桩框架在实现指定功能时，插桩的代码和程序原始代码是相对独立，互不影响的。

对于基本块的搜索，要能够无一遗漏地搜索识别出所有的基本块。基本块的分割是由分支结构来决定的，但并不是所有基本块都是由跳转指令（在ARM指令集中即B类指令、LDR PC,xxx类指令、返回类指令）作为出口和入口的，例如在汇编代码里常见的Jump table类型的跳转，或者是C++虚函数中用到的vtable类型的跳转等，均不是使用跳转指令作为基本块的分界点的，但内部都包含了多个基本块。基本块的搜索可以算是整个Fuzzing反馈的重中之重，因为基本块是来计算代码执行的覆盖率的关键，有错漏的话会直接影响代码覆盖率的计算。

对于与AFL对接部分，本项目作为AFL静态插桩功能的扩展，需要与AFL对接，在汇编层面上实现AFL原本的setup、fork-server、log 三个功能。关于setup功能，需要与AFL主进程建立通信管道，包括状态管道和命令管道，分别用来传输反馈数据和执行命令，同时还要建立好共享内存区，用来构建一个哈希表来记录执行路径，并做为计算覆盖率的标准。而在fork-server部分，在第一次执

行该程序时，所有动态库都加载完成之后，该进程就要成为一个fork-server，类似于一个只有2个状态的状态机，状态一为等待状态，等待命令管道发送执行命令，然后进入状态二，即fork状态，fork出新的子进程，然后通过状态管道返回给AFL主进程对应的进程信息，之后重新返回状态一，等待新的命令。fork-server能有效提高Fuzzing效率，因为减少了每一次执行都重新加载动态库以及初始化程序空间的时间，直接从主要逻辑开始执行，加载以及初始化工作会且仅会执行一次，而fork作为一个高效率的系统调用，在以虚拟内存作为模型的类Unix系统中，实现了写时拷贝的语义，并不实际复制整块物理内存，通过仅仅拷贝页表，使得虚拟内存都指向同一物理内存，而在其对应的虚拟内存页被写操作时，才对该页进行复制拷贝，这使得fork的代价远小于重新运行程序代价。而最后的log部分，则是需要进行一次hash操作，把当前程序位置以及上一程序位置进行哈希后，在setup时建立的哈希表中进行一次hit记录，而当前程序位置，是由一个在插桩时给每一个位置插入的magic数来定义的，这有效减少了一定的哈希碰撞。

2.2 整体系统结构

2.2.1 静态插桩框架结构

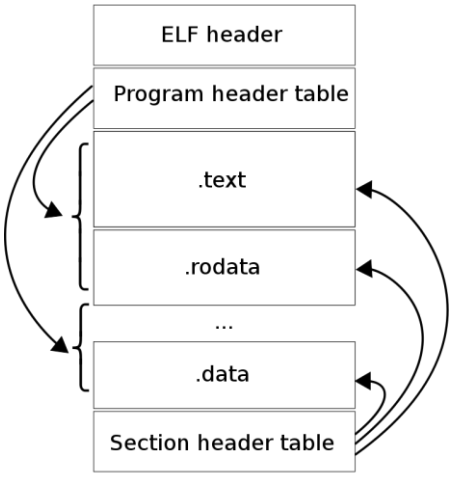

图2-1 ELF文件基本格式

静态插桩框架的对外接口为一个ELF类，其构造函数接受一个ELF文件名作为参数，加载并解析对应的ELF文件，简略的ELF文件格式如图2-1。之后对该ELF

的所有操作都是通过该ELF类提供的接口来实现的，该ELF类的UML类图如图2-2所示。

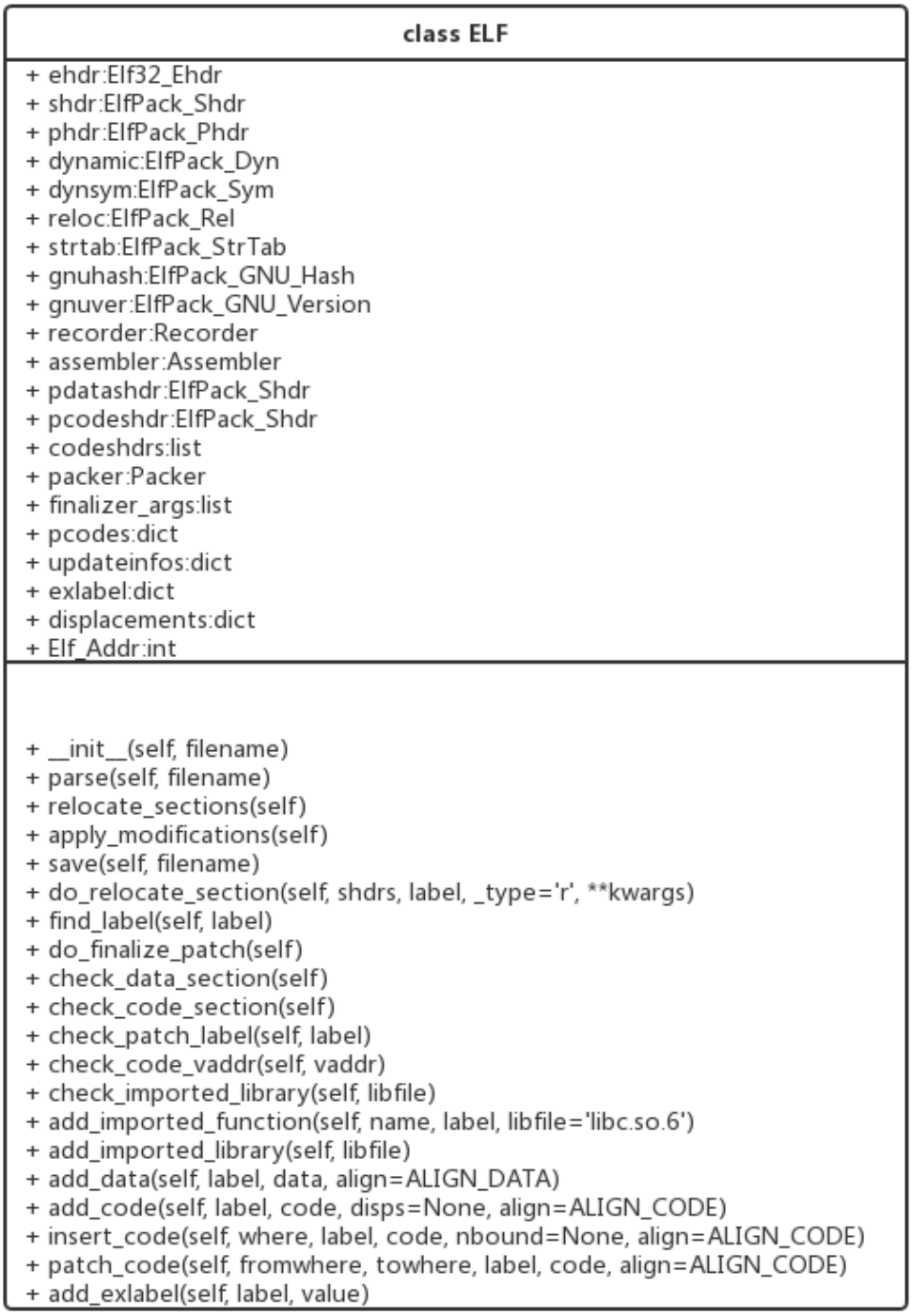

图2-2 静态插桩框架ELF类的UML类图

除了ELF类之外，本项目实现的静态插桩框架还包括了Elf_Ehdr、ElfPack_Shdr、ElfPack_Dyn、ElfPack_Sym等多个内部类来保证了ELF类工作正常、开发逻辑清晰以及调试方便。这些类的引用关系以及作用如图2-3所示。对于后续使用该静态插桩框架进行插桩工作的程序而言，只需要对ELF类操作之后，调用ELF.save()函数，就可以把修改后的程序，重新打包成ARM架构可执行二进制ELF文件，可以正常执行。

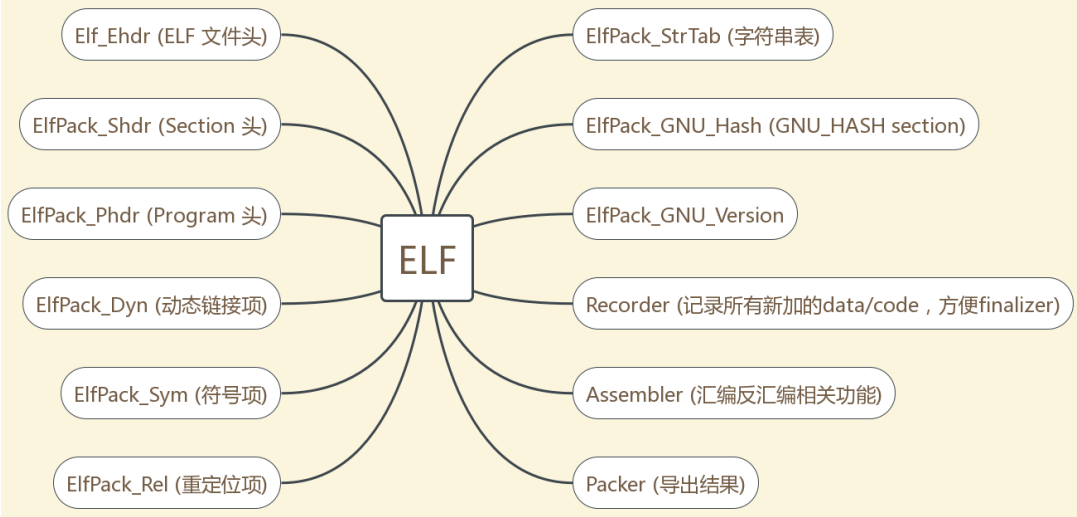

图2-3 静态插桩框架ELF类引用的内部类及作用

2.2.2 与AFL逻辑相连接的逻辑架构

AFL在Fuzzing时，简要的工作架构如图2-4所示。在进行Fuzzing的时候，存在3类主体：

1. AFL主进程（Fuzzer）。作为AFL的主进程，负责启动AFL进程，启动需要Fuzzing的程序进程，按着一定规律和Fork Server的反馈信息来变异输入数据，并通过2个管道与Fork Server沟通传递命令和反馈信息，以及通过一定的UI（可省略）反馈给用户。该类主体对于一次Fuzzing有且仅有一个，即AFL主进程。

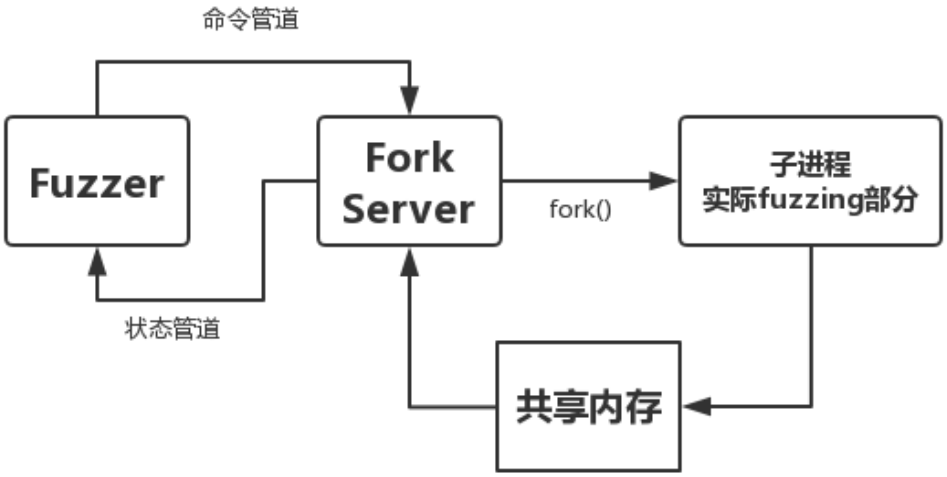

图2-4 AFL在Fuzzing时的工作架构

2. Fork Server。该类主体对于一次Fuzzing也有且仅有一个，事实上，Fork Server便是AFL主进程第一次启动的需要Fuzzing的程序进程。在Fork Server被AFL主进程启动之后，便在等待状态和执行状态中不断切换。每当收到一条来自AFL主进程的命令，便进行一次fork()操作，然后把一些信息通过状态管道传输给AFL主进程，接着继续进入等待状态中，等待下一次命令到来。
3. Fork Server的子进程。这些子进程才是实际对程序的各个分支代码进行Fuzzing的部分，执行被Fuzzing程序的代码，记录分支的执行次数和跳转情况，并反馈在子进程和Fork Server共享的一块内存里。如果产生了crash，代表可能当前输入导致该程序的某个分支出现了bug。

2.2.3 搜索程序基本块

该部分的程序架构较为简单，如图2-5所示。首先先调用IDA API获取程序的所有函数，然后对所有函数向上溯源，找到其最根部调用的函数。如果其根部调用是main函数，代表该函数是程序正常执行逻辑里，有可能会执行到的，需要进行Fuzzing，便把这些函数设上标记，表示其是需要进行插桩的，之后对所有需要进行插桩的函数调用IDA API进行搜索，找到该函数范围内的所有基本块。然后对这些基本块进行去重，以及对其属性进行判断，包括基本块所使用

的指令集类型，基本块起止地址，基本块大小。最后把这些信息以Json格式保存起来，方便后续的插桩。

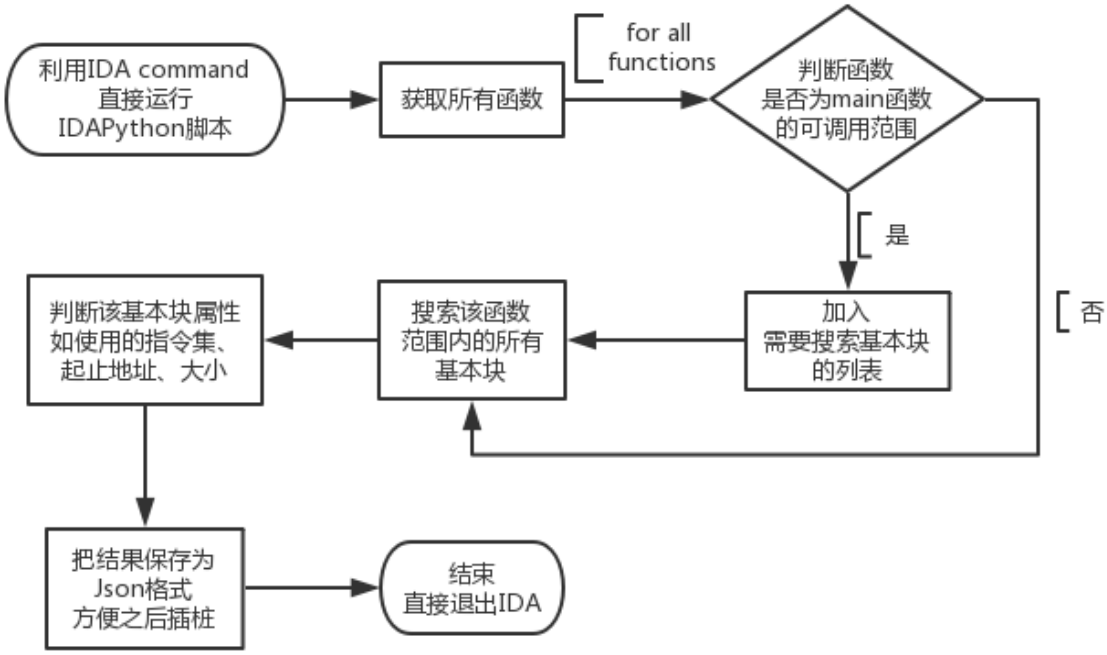

图2-5 搜索程序基本块的程序流程图

2.2.4 项目整体逻辑架构

为了更直观简洁的说明本项目的执行逻辑流程，以图2-6的程序流程图来进行说明整体的执行逻辑。程序一开始接受一个ARM架构的ELF程序文件，调用IDA Command的接口执行搜索基本块的IDAPython脚本，以Json格式输出搜索到的基本块信息。之后进行插桩部分的逻辑，读取上一步获得的Json格式的基本块信息，对所有基本块的开头进行插桩，插入符合AFL逻辑的代码，并重新打包成ELF文件。最后，即可使用AFL，对插桩后的ELF文件进行Fuzzing工作。

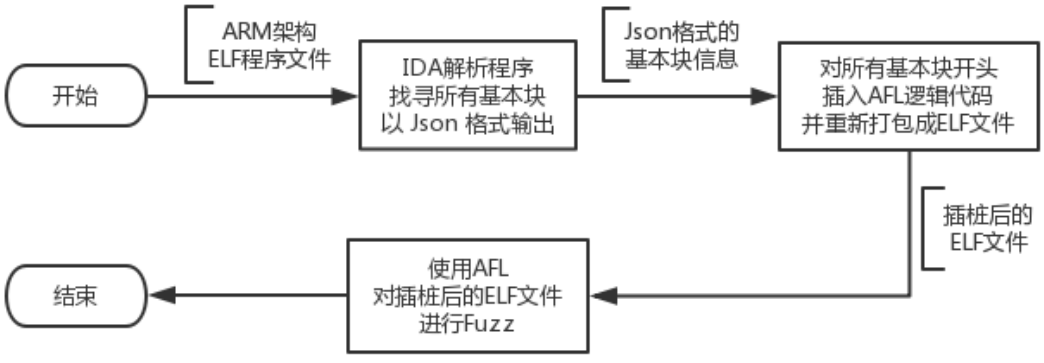

图2-6 本项目整体逻辑架构

2.3 关键技术分析

2.3.1 静态插桩

在一开始尝试使用一些前人已经完成的框架（如LIEF、elfio等），来进行插桩的工作，然而这些开源架构并不足够满足本项目对静态插桩框架的需求，对ARM架构支持不够，功能较少，只支持替换指定文件偏移的一段代码，但本项目需要插入大量更长的代码用来执行AFL的逻辑，插入新数据段来存储AFL的全局变量，以及需要新增外部导入函数，这些都是现有框架不包括的功能。所以本项目决定从头开始，开发一套支持ARM架构ELF程序文件的静态插桩框架。

该静态插桩框架的插桩后生成的ELF文件符合GNU标准，其解析原始ELF文件，为该ELF文件插桩后打包。同时，为了开发简洁以及更好的鲁棒性，该静态插桩框架把所有新加入的代码都统一加入一个新段pcode，把所有新加入的的数据都统一加入一个新段pdata，尽量保证了原始文件部分被改动部分尽量少，防止原始代码结构变得过于复杂以及减少调试的难度。

2.3.2 基本块的搜索

一开始，本项目想尝试使用一些现有的框架或其组合去完成搜索基本块的工作，因为这样整体项目会比较轻量，主要尝试了以下几种：

1. Miasm[12]。 Miasm是一个开源的python逆向工程框架，在提供一个函数地址的前提下，能够搜索该函数内部的基本块。在尝试后，发现该框架找出基本块地址与实际基本块地址，有可能会存在一定偏移，对于大的基本块来说没有区别，但对较小的基本块就是致命错误，所以放弃了这个方案。
2. Barf[13]。 Barf也是一个开源的二进制分析以及逆向框架，和Miasm一样，在给定函数地址的前提下，可以搜索该函数内部的基本块。初步测试的时候效果不错，于是又使用了另一个Sibyl框架（Sibyl是基于Miasm2的开发的一个函数分析框架），来获取所有函数的位置，之后用Barf对所有函数进行分析，搜索基本块。然而在后续测试时候发现，Barf

对以Jump table为汇编结果的switch语句支持很差，会漏掉很多基本块。

3. Radare2。 Radare2是一个开源的逆向和二进制分析框架，与IDA类似，但相对轻量。功能强大，包括反汇编、函数分析、打补丁等等，同时也具备脚本加载能力。然而，在测试中，虽然比barf框架好了很多，但其找到的基本块也有所缺漏。

在多番尝试，却依然没法找全、找准所有基本块后后，本项目最后使用了IDA API来完成搜索基本块的工作。IDA是一个多平台支持的反汇编器和调试器，它提供与反汇编、程序分析、调试相关的非常多的功能，而且作为一个商业项目，其功能性和准确性很高，但其体量较大，这也是一开始选择轻量级的开源框架而不是IDA的原因。使用IDA API编写了搜索基本块的脚本并测试后，其找到的数量和准确性是试验的几个框架中最高的，所以最终使用IDA API来搜索基本块。

2.3.3 AFL Fuzzing

之所以选择AFL进行ARM架构的静态插桩功能扩展，是因为AFL作为十分经典，在当今又最为流行的Fuzzing框架，有着较为简洁的代码架构，功能又十分全面、高效，成为很多安全从业人员喜爱的模糊测试引擎。

2.4 本章小结

本章对本项目的架构进行了梳理,对各个主要部分都进行了图文并茂的阐述，也对整体架构进行了简单的描述，并且也解释了一些开发时候使用的框架和技术，以及使用的原因或是选择自己开发的原因。

3 项目主要工作内容

3.1 AFL 插桩代码

3.1.1 需求分析

由于本项目是基于AFL的功能扩展，所以需要和AFL原始代码进行通信，这就要保证本项目对二进制文件插的桩，在逻辑上与AFL是互通的。所以插入的桩必须要从汇编层面上，完成三件事：

1. afl_setup

   进行一些必要的错误检查，然后获取环境变量的内容（主进程保存的共享内存的标志符）并将其存入全局变量。然后，通过调用shmat()，将这块共享内存映射到了自己的虚拟内存空间中，并将其地址也保存在全局变量中。由此，便完成了AFL的子进程的初始化工作。

2. afl_forkserver

   首先，Fork server需要将alive信息写入状态管道，来通知Fuzzer，已经准备完毕，可以开始fork了。接下来，Fork server进入等待状态，读取命令管道，直到Fuzzer通知其开始fork。一旦Fork server接收到Fuzzer的信息，便调用fork()，得到父进程和子进程。子进程是实际执行二进制目标程序的进程，其需要关闭不再需要的管道，并继续执行。而父进程则仍然作为Fork server运行。这时，父进程需要将子进程的进程号通过状态管道发送给Fuzzer，并等待子进程执行完毕。一旦子进程执行完毕，则再通过状态管道，将其结束状态发送给Fuzzer。之后再次进入等待状态，等待下一个fork命令的到来。

3. afl_maybe_log

   这一部分的插桩代码主要负责记录程序执行的覆盖率的工作，这就要求插桩的代码需要知道当前PC的位置，所以在进行插桩时，需要给每一个插桩位置一个独一无二的“标识”。在这一部分中，由于父进程和子进程是通过共享内存进行数据交换的，所以需要把当前的执行路径记录在这块共享内存上，才能由Fuzzer计算程序执行的覆盖率。

3.1.2 架构设计

本项目中，需要对目标程序进行Fuzzing，并与AFL的Fuzzer主进程进行通信，插桩程序执行以及与AFL主进程通信流程，如图3-1所示。

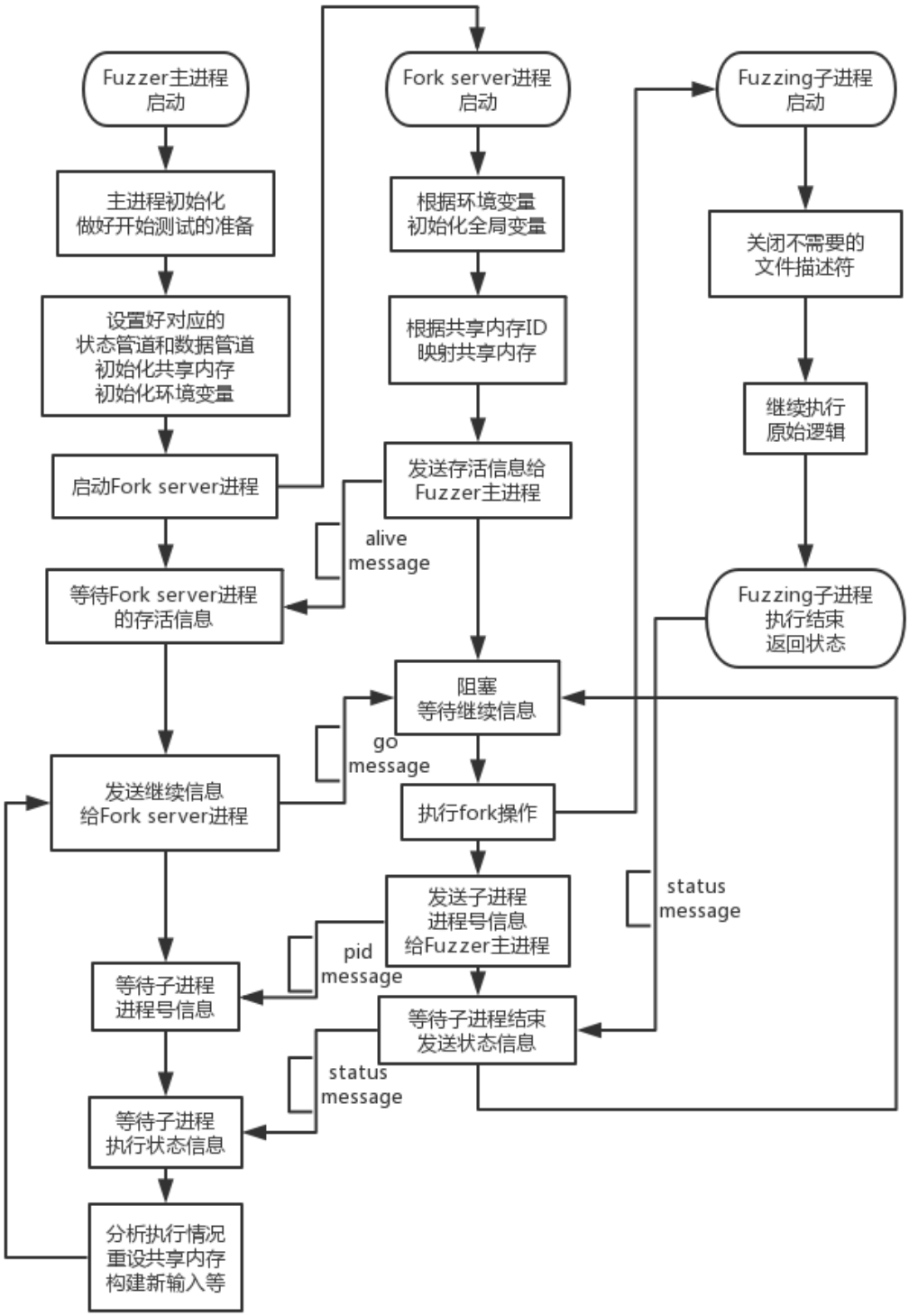

图3-1 插桩程序执行以及与AFL主进程通信流程

3.1.3 实现细节

对于本项目给二进制文件插入的桩有两种，第一种是初始化相关的，也就是afl_setup以及afl_forkserver的代码。这两段代码只会插入在程序的入口点处，即main函数的第一个基本块的开始。当然，作为一个基本块，其依然要记录自己的路径信息，所以在程序入口处会如代码3-1所示地插入代码。

```
function main(void)
    afl_setup();
    afl_forkserver();
    afl_maybe_log();
    // Original code
    ......
```

代码3-1 初始化时插入的桩的伪代码及其位置

而对于非程序入口点，不需要插入初始化代码，只需要进行路径记录，所以只会插入afl_maybe_log的代码，如代码3-2所示。

```
function foo(void)
    afl_maybe_log();
    // Original code
    ......
```

代码3-2 非程序入口点插入的桩的伪代码及其位置

对于afl_setup，其需要完成取环境变量并设置全局变量，以及初始化共享内存的工作，其伪代码代码3-3所示，先取出环境变量中__AFL_SHM_ID（代表共享内存ID）、AFL_INST_RATIO（根据AFL文档提到，对于过于庞大复杂的程序，需要减少一定比例的路径记录量，来防止用于记录路径的哈希表产生碰撞）的值，如果环境变量AFL_INST_RATIO被设置，那全局变量afl_inst_rms也会被设置，用来在每次记录的时候，用作概率（假设随机数的分布是随机的），来判断是否应该进行路径的记录。而环境变量__AFL_SHM_ID则是又Fuzzer设置的，一定存在，用作共享内存的ID。然后以获取到的共享内存ID作为参数，调用shmat()来获取共享内存，默认大小为1 << 16字节，即64k。该共享内存实际上由Fuzzer、Fork server、fuzzing子进程共享，用来传递fuzzing子进程的路径信息。Fuzzer在调用Fork server的时候设置好__AFL_SHM_ID环境变量，这样就能往其子进

程，也就是Fork server传递共享内存的ID。而fuzzing子进程由于afl_setup时存储了共享内存的地址，所以直接向那片地址写路径信息即可。

```
function afl_setup (void)
    Get the environment variables of "__AFL_SHM_ID" and "AFL_INST_RATIO", and set them to global variables.
    if AFL_INST_RATIO is set then
        afl_inst_rms ← 1 << 16 * AFL_INST_RATIO / 100
    end if
    shm_id ← __AFL_SHM_ID
    global variable afl_area_ptr ← shmat(shm_id, NULL, 0)
end function
```

代码3-3 afl_setup实现伪代码

在初始化之后，便要进行Fork server的工作，其伪代码代码3-4所示。首先，会对共享内存指针进行检查，来防止未初始化就进行Fuzzing，然后通过状态管道，想其父进程Fuzzer发送存活信息，表示Fork server已经准备好进行Fork工作，接着便进入一个死循环。在这个死循环中，Fork server阻塞地等待Fuzzer发来的继续信息，当收到继续信息，便开始fork()操作。对于子进程而言，即将进行的是原始程序的逻辑，而且所有通讯都以共享内存的形式进行，所以把状态管道和命令管道都关闭了。而对于父进程，即Fork server进程，需要把子进程的进程号通过状态管道传给Fuzzer，然后便是等待fuzzing子进程运行结束。在其结束之后，把fuzzing子进程的运行结果，也就是waitpid返回的状态，通过状态管道发送给Fuzzer，由Fuzzer判断程序是否正常运行或者产生了crash，并把结果记录。最后重新回到循环，继续等待Fuzzer发来的继续信息后进入下一次fork操作。

```
function afl_forkserver (void)
    if global variable afl_area_ptr == Null then
        return error
    end if
    if write alive messge to status fd failed then
        return error
    end if
    global variable afl_forksrv_pid ← getpid()
    while True do
        Waitting for go message from fuzzer, blocked.
        child_pid ← fork()
```

```
        if fork failed then
            return error
        end if
        if in child process then
            afl_fork_child ← 1
            close data fd
            close status fd
            return good
        end if
        if in parent process then
            if wirte child_pid to status fd failed then
                return error
            end if
            waiting for child process return
            status ← child process return value
            if wirte status to status fd failed then
                return error
            end if
        end if
end function
```

代码3-4 afl_forkserver实现伪代码

对于fuzzing子进程所而言，需要执行的仅仅只有afl_maybe_log，其伪代码代码3-5所示。首先，afl_maybe_log也会对共享内存指针进行检查，之后进行特定的哈希函数，来把当前位置和上一位置以及进行哈希，这样就可以把路径信息包含其中。然后判断了*cur_loc*和*afl_inst_rm*的大小，*afl_inst_rm*是在alf_setup阶段设置的一个控制插桩记录代码执行数量的变量，用来防止过于复杂的程序的过多跳转使得发生哈希碰撞。如果*cur_loc*比*afl_inst_rm*小，那么便在共享内存里进行记录。之后把当前位置右移了一位存入*prev_loc*。这是为了区分跳转顺序，例如存在跳转A -> B、B -> A，在没有右移存在的情况下，无法区分，因为异或操作是符合交换律的。同时，右移也可以区分递归调用，例如存在跳转A -> A、B -> B，在没有右移的情况下，两者的异或结果会都变成0，导致无法区分。

而*prev_loc*变量，被设定成了线程局部的变量，存在TLS（Thread Local Storage 线程局部存储）里，这是为了支持对多线程程序进行Fuzzing。如果*prev_loc*是一个全局变量，当多线程程序运行到afl_maybe_log时候，可能会因为

线程调度的原因，被其他线程抢占，而抢占的线程如果也运行到了afl_maybe_log，而*prev_loc*还是上一线程执行的位置，着就会使得记录下来的路劲信息出现错误，所以*prev_loc*变量必须是一个线程局部的变量。

在这里的*cur_loc*和*prev_loc*两个局部变量指的都是当前位置的标识符，这是在程序进行插桩时，硬编码进程序里的一个magic，这个magic由当前PC计算出，在本项目中，直接使用了CRC32(PC)来计算和当前PC相对应的magic。CRC32算法是一个常见的校验算法，是对一段数据进行循环校验的散列函数，输出一个32位整数，在本项目中用来以当前PC计算每一个插桩点的标识magic（32位整数）。

```
function afl_maybe_log (void)
    define thread-local variable prev_loc
    if global variable afl_area_ptr == Null then
        return error
    end if
    cur_loc ← (cur_loc >> 4) ^ (cur_loc << 8)
    cur_loc ← cur_loc & (1 << 16 – 1)
    if cur_loc >= afl_inst_rms then
        return good
    end if
    position ← cur_loc ^ prev_loc
    global variable afl_area_ptr[position] ← global variable afl_area_ptr[position] + 1
    prev_loc = cur_loc >> 1
end function
```

代码3-5 afl_maybe_log实现伪代码

由于在fuzzing子进程运行过程中，Fork server进程被阻塞在waitpid()里等待其返回，所以在fuzzing子进程运行结束后，会把当前状态发送给Fork server进程，再由Fork server进程把其执行状态发送给Fuzzer主进程。

而Fuzzer主进程在收到之后，会分析其返回状态（正常返回、异常退出或crash）和其执行路径（也就是共享内存里的哈希表），这部分不在本项目的考虑之列。

3.2 对二进制文件进行插桩

3.2.1 需求分析

在本项目中，需要实现对ARM的ELF文件的静态插桩框架，我们要向已经编译完成的二进制文件中插入代码和数据，所以静态插桩框架必须支持这2大功能。而由于原始编译的二进制文件的每个段都是页对齐的，大小有限，而我们需要插入大量代码，所以需要对二进制文件进行加段操作，添加新的段来存放我们需要插入的代码和数据，并设置好相应属性。

对于插入代码部分，由于如果直接对代码段的某个位置插入二进制代码，再后移之后的代码，这将会导致整个程序的结构混乱，所有跨过插桩位置的相对寻址全部错误，以及在程序加载时的重定位工作也会发生错误。所以选择了只把当前位置的代码改为跳转代码，跳转到新增的代码段，然后在新增的代码段里重新执行原始被修改的那部分代码，接着再插入实际我们需要插入的代码。在执行完毕之后再跳回原始代码。

而对于插入数据，这就相对简单的多，只要把所有新增的数据都放在新增的数据段中，然后在代码里做相对寻址的引用就好了。

除此之外，需要找到桩的插入点，也就是基本块的地址，同时也要标注好插入点使用的指令集是ARM指令集还是Thumb指令集。

在进行实际插桩时，需要保存好插桩前后的上下文，保证插桩代码和原始程序代码的独立性。同时，对于被原始程序中，被替换的代码是PC-related的需要特殊对待，来保证执行该被替换代码的正确性。

3.2.2 架构设计

在本项目中静态插桩框架对外的接口全部写在了ELF类中，其主要功能函数的定义以及完成的功能点如下，用来满足上述提到的需求：

- parse(self, filename)：解析一个ARM架构的ELF文件，同时设定ELF类的各个成员变量，在ELF类初始化时用到。

- save(self, filename)：保存修改，并存储修改过后的程序为ARM架构的ELF文件格式。
- add_imported_function(self, name, label, libfile='libc.so.6')：新增导入函数，从已有库*libfile*中添加名为*name*的函数，该函数在本程序中的名字为*lable*。
- add_imported_library(self, libfile)：新增名为*libfile*的导入库。
- add_data(self, label, data, align=ALIGN_DATA)：新增一个数据项，该数据项在本程序中名为*label*，初始化数据为*data*，*align*参数表示该数据项是否对齐，以及对齐的字节大小。这个函数仅仅只添加一个新数据项而并没有对该数据段进行引用。
- add_code(self, label, code, disps=None, align=ALIGN_CODE)：新增一段代码名为*label*，该段代码实际汇编指令由*code*给出，*align*参数意义同上。这个函数也仅仅只添加一段新代码而并没有对该代码进行引用。
- insert_code(self, where, label, code, nbound=None, align=ALIGN_CODE)：在指定位置*where*插入一段名为*lable*的代码，*where*参数指向的是虚拟地址偏移而非文件偏移，*code*、*align*参数意义同上。该函数实际上已经实际上修改了原始程序的执行逻辑，是在add_code函数的基础上，添加了从*where*到*lable*指向的新代码的跳转。
- patch_code(self, fromwhere, towhere, label, code, align=ALIGN_CODE)：该函数不进行跳转，直接抹除原来fromwhere到towhere之间的所有代码，替换成一段名为lable的新代码。

3.2.3 实现细节

静态插桩框架部分，主要有3大功能，首先是新增数据项功能。如代码3-6所示，在初次增加新数据项时，会以RW权限新增一个数据段，名为*.pdata*，加入新数据段时，会同时在strtab（ELF中用来存放所有字符串的表）以及section header（ELF中用来记录所有section信息的表）更新该新数据段的信息。在初始化完新添加的数据段或是*.pdata*已经存在之后，则会添加该新数据项的名字

到strtab中，并以一定的对齐方式加入到新添加的*.pdata*的最后，同时也会记录下对应的偏移和长度。

```
function add_data(self, label, data, align=ALIGN_DATA)
    if .pdata haven't be initialized then
        add '.pdata' to strtab
        add new .pdata section to section header
    end if
    if label has exist in .pada section then
        return error
    end if
    add label to items of .pdada section
    align data to align
    recode the length of data and the offset that it will be in .pada section
    append data to the end of .pada section
end function
```

代码3-6 add_data实现伪代码

其次是实现插入新代码功能insert_code函数，如代码3-7所示，在初次增加新代码时，会以RX权限新增一个代码段，名为*.pcode*，加入新代码段时，会同时在strtab以及section header更新该新代码段的信息。在初始化完新添加的代码段或是*.pcode*段已经存在之后，会对需要插入的代码进行包装，使得插桩代码能和原始代码保持相对独立，同时也能在最开始执行需要插入位置被替换的原始代码。在包装过程中，主要就是完成跳转以及原始指令的执行，但是需要考虑原始代码的可执行性，因为在从原始位置跳转到新代码段时，PC改变了，这会使得所有PC-related的指令的执行出现错误，所以需要修改几乎所有PC-related的指令代码，除了LDR PC, [...](, ...)?，以及pop {pc}，因为这两类指令的执行结果，与PC当前值并没有关系。

对于需要插入位置的代码是一个调用jump table跳转的位置，由于jump table的位置都是PC-related的，所以需要特殊包装，这主要包括ldrls pc, [pc,xxx]指令，对于这类指令，需要从当前位置向上搜索最近的cmp指令，找到jump table的大小，然后把整个jump table搬到新代码段这条指令的后面。除此之外，就剩下PC-related的运算指令或是PC-load指令，包括add reg, pc, (reg|#number)和ldr reg, [pc(, #x)?]，对于这类指令，先在R0-R4中选择一

个没在这条指令中用到的寄存器，把它存在栈上，然后把当前PC存入该寄存器，最后把这条指令中的PC都替换成这个寄存器就可以了。

在指令包装完毕之后，需要在插桩代码的前后分别加上保存和恢复上下文的代码，然后把这段新代码插入的新增的.pcode代码段的末尾，并把原始位置的代码替换成跳转到新增代码的B指令即可。

在插桩完成后，其代码执行逻辑将会变成如图3-2所示。

```
function insert_code(self, where, label, code, nbound=None, align=ALIGN_CODE)
    if .pcode haven't be initialized then
        add '.pcode' to strtab
        add new .pdata section to section header
    end if
    if label has exist in .pada section then
        return error
    end if
    add label to items of .pcode section
    if code is PC-related code then
        if code is jump table code then
            copy whole jump table after new code
        end if
        if code is PC-load or PC-arithmetic code then
            find free register in R0-R4 and store it in stack
            free register ← PC
            replace PC in code with free register
            restore free register from stack
        end if
    end if
    add jump back code to the end of code
    add warped code to .pcode section
    patch original code position to a jump to new code
end function
```

代码3-7 insert_code实现伪代码

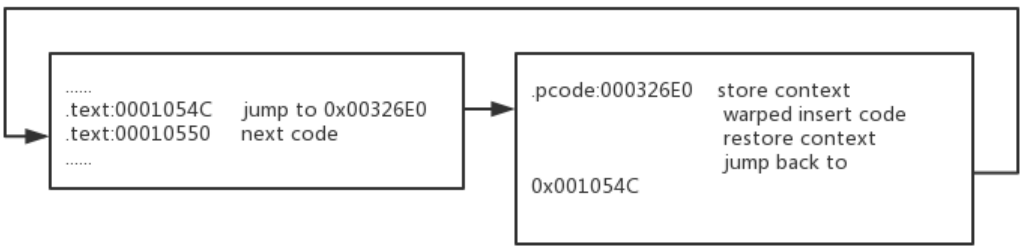

图3-2 insert_code插桩后代码的执行流程

插桩框架的第三个主要功能是添加库函数功能，add_imported_function。该函数能够从外部库中导入一个库函数到该ELF文件中，如果需要导入的外部库函数所在的外部库并没被该ELF文件导入，则会先导入对应库。如代码3-8所示，首先检查导入库的库函数所在的库是否已经导入，如果没有则要先导入该库。然后会向strtab中添加需要加入的函数*label*，继续再在程序的.dynsym中加入这个函数，表明其需要动态链接，接着添加一个新数据项，以及在重定位项中加入该函数，用以重定位。

```
function  add_imported_function(self, name, label, libfile='libc.so.6')
    if libfile hasn't been imported then
        import libfile
    end if
    add name to strtab
    add function need import to .dynsym section
    add a point to this function for relocation
    add function to relocation item
    add item in gnu section
end function
```

代码3-8 add_imported_function实现伪代码

当具体进行插桩时，需要先添加好所有AFL代码里需要的全局变量，然后对搜索到的所有基本块的最开始进行insert_code操作，插入对应的AFL代码即可。

3.3 本章小结

本章将对本项目主体部分做了详细的需求分析，并且对其实现的具体细节主要通过图文和伪代码的形式，做了详细阐述。包括了AFL对接逻辑代码的执行流程和实现细节，以及静态插桩框架的原理和主要函数的具体细节。

4 测试及结果

4.1 对于搜索基本块程序的测试

将本项目从二进制程序搜索到的基本块，与AFL从LLVM层搜索到的基本块的数量进行对比。对于一个简单的带有判断以及函数的程序，本项目搜索到了9个基本块，而AFL搜索到了8个基本块。对于一个复杂的开源程序who，本项目搜索到了2087个基本块，而AFL搜索到了2150个基本块。

由于两者搜索时面向的对象的不同，AFL是从LLVM层搜索基本块，而本项目则是从编译完成后的二进制程序中进行搜索，由于编译器会对一些汇编代码进行合并、分割和优化，所以表现出的基本块数量有一定差别是正常的。例如，对于switch语句的处理，编译器会优化成一个jump table形式，而AFL由于是在插完桩之后编译器才进行优化，使得其没有被识别成一个jump table，而是多次判断跳转的形似，造成了基本块数量上的差异。

对于测试工作量较小的简单程序，通过反汇编插桩后的代码，除了由于编译器优化造成的差异以外，可以从逻辑上看出二者插桩的位置实际上是一样的。

4.2 对于静态插桩框架的测试

在本项目中，静态插桩框架会在所有基本块的最开始，插入代码，保存上下文信息之后，跳转到插入的AFL代码里，进行记录以及和AFL主进程沟通的工作。

以一个简单程序为例，将插桩前的代码反汇编后，取main函数的一个片段，可以看出其中包含了多个基本块，如图4-1所示。在图4-1中，包含了多个基本块，在插桩之后，会在所有基本块的开头加入跳转代码，跳到新添加的一个函数中，先保存上下文后，再按要求依次调用AFL的代码，来与AFL进行沟通，之后执行原始被替换的代码，最后跳转回原始PC。插桩后的该片段如图4-2所示。

```
105bc:       e59f0044        ldr     r0, [pc, #68]   ; 10608 <main+0xbc>
105c0:       ebffff7c        bl      103b8 <puts@plt>
105c4:       ea000007        b       105e8 <main+0x9c>
105c8:       ebffffd3        bl      1051c <foo>
105cc:       ea000005        b       105e8 <main+0x9c>
105d0:       e59f0030        ldr     r0, [pc, #48]   ; 10608 <main+0xbc>
105d4:       ebffff77        bl      103b8 <puts@plt>
105d8:       ea000002        b       105e8 <main+0x9c>
105dc:       e59f0028        ldr     r0, [pc, #40]   ; 1060c <main+0xc0>
105e0:       ebffff74        bl      103b8 <puts@plt>
105e4:       e1a00000        nop                     ; (mov r0, r0)
105e8:       e1a00000        nop                     ; (mov r0, r0)
105ec:       e1a00000        nop                     ; (mov r0, r0)
105f0:       e1a00003        mov     r0, r3
105f4:       e24bd004        sub     sp, fp, #4
105f8:       e8bd8800        pop     {fp, pc}
```

图4-1 插桩前main函数中的一个片段

```
105bc:       ea00886f        b       32780 <__bss_end__+0x11744>
105c0:       ebffff7c        bl      103b8 <puts@plt>
105c4:       ea000007        b       105e8 <main+0x9c>
105c8:       ea008884        b       327e0 <__bss_end__+0x117a4>
105cc:       ea000005        b       105e8 <main+0x9c>
105d0:       ea008896        b       32830 <__bss_end__+0x117f4>
105d4:       ebffff77        bl      103b8 <puts@plt>
105d8:       ea000002        b       105e8 <main+0x9c>
105dc:       ea0088ab        b       32890 <__bss_end__+0x11854>
105e0:       ebffff74        bl      103b8 <puts@plt>
105e4:       e1a00000        nop                     ; (mov r0, r0)
105e8:       ea0088c0        b       328f0 <__bss_end__+0x118b4>
105ec:       e1a00000        nop                     ; (mov r0, r0)
105f0:       e1a00003        mov     r0, r3
105f4:       e24bd004        sub     sp, fp, #4
105f8:       e8bd8800        pop     {fp, pc}
```

图4-2 插桩后main函数中的一个片段

在图4-2中，红框里是插桩的代码，由于每个基本块插桩需要做的事都不同，所以跳转的位置也都不同，跳转之后主要做了五件事,如图4-3所示：

1) 保存上下文

2) 跳转执行AFL代码

3) 恢复上下文

4) 重新执行被插桩代码覆盖的代码

5) 跳转回之前的地址

```
.pcode:000326E0    STMFD   SP!, {R0-R12,LR}
.pcode:000326E4    SUB     SP, SP, #0xC
.pcode:000326E8    MRS     R0, CPSR
.pcode:000326EC    STR     R0, [SP,#0x44+var_3C]
.pcode:000326F0    LDR     R0, =0xF3257879
.pcode:000326F4    STR     R0, [SP,#0x44+var_40]
.pcode:000326F8    STR     SP, [SP,#0x44+var_44]
.pcode:000326FC    MOV     R0, SP
.pcode:00032700    LDR     R3, =hook_0
.pcode:00032704    BLX     R3
.pcode:00032708    LDR     R0, [SP,#0x44+var_3C]
.pcode:0003270C    ADD     SP, SP, #0xC
.pcode:00032710    MSR     CPSR_cf, R0
.pcode:00032714    LDMFD   SP!, {R0-R12,LR}
.pcode:00032718    STMFD   SP!, {R11,LR}
.pcode:0003271C    B       loc_10550
```

图4-3 插入的每个桩的实际代码

在图4-3中，（按从上到下的顺序）红色部分负责保存上下文，天蓝色部分调用AFL代码，深蓝色部分恢复上下文，绿色部分执行被插桩代码覆盖的代码，紫色部分重新跳回原始PC，完成了整个插桩后跳转流程，同时保证了上下文的统一，以及插入代码与原始代码的独立性，防止跳转回去之后执行失败。

4.3 结合 AFL 的整体测试

结合AFL，对一个简单的带有bug的程序进行了测试，如图4-4所示，顺利跑出了crash，经过验证确实是能够造成程序崩溃的crash。

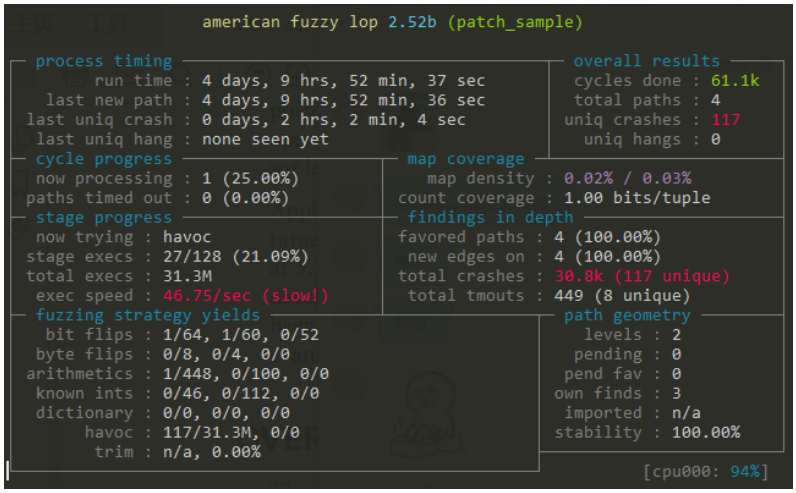

图4-4 对简单带bug程序的测试结果

除此之外，还分别进行了本项目针对开源程序who编译所得的二进制程序以及AFL编译的who在ARM平台上的测试，作为对比。如图4-4所示，为本项目二进制程序who的运行状态，可以看出其能够和AFL沟通，并正常运行。而如图4-5所示，是AFL编译的who在与本项目插桩的who，提供了相同的输入，运行了相似次数后的运行状态图。

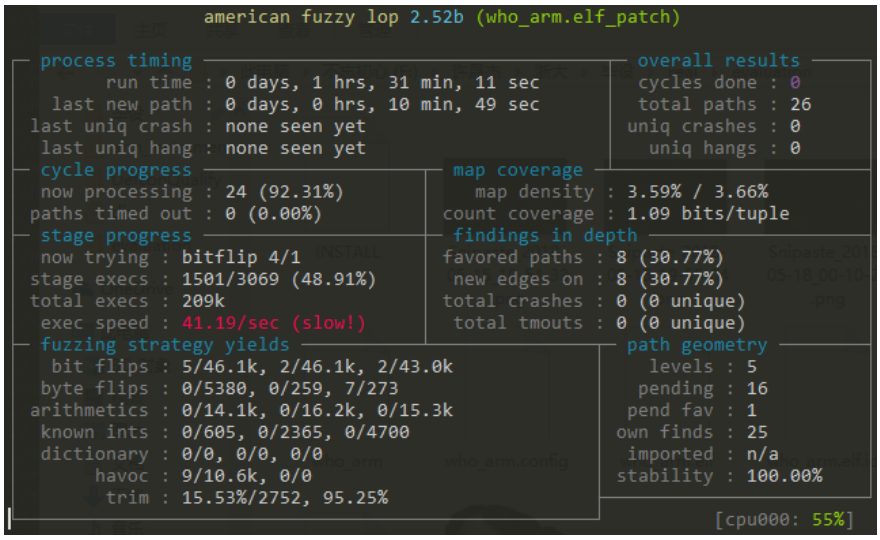

图4-4 本项目结合AFL的运行状态

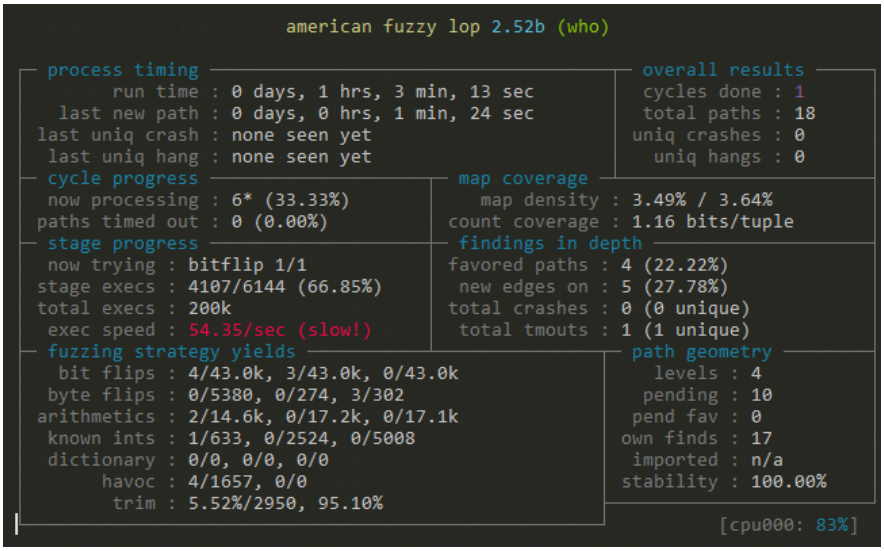

图4-5 AFL编译的程序的运行状态

可以看出，两者的map density，即共享内存的密度，基本一致，代表两者在相同输入，相似执行次数的条件下，再加上上一节提到的两者插桩的基本块

数量相近，所Fuzzing的程序的执行分支覆盖率相似。可以基本确定本项目的程序设计和实现没有问题。

而速度方面，由于本身本项目进行插桩的代码量就比原始AFL代码插桩多了几十行汇编代码，再加上本项目插桩的所有点都添加了2个跳转用来进如新代码段和返回原始代码位置，这可能会造成cache的的频繁进出操作，造成了本项目的Fuzzing执行速度会略小于原始AFL执行速度。在对于who程序的测试上，在两者都执行了200k次fuzzing子进程情况下，原始AFL程序的执行速度约为57.73exec/sec，而本项目进行插桩的程序的执行速度约为39.70exec/sec。

4.4 本章小结

本章主要是对本项目的成果分别进行了测试，包括静态插桩框架的插桩结果，基本块的搜索，结合AFL运行的结果。同时，也对测试结果做出了分析，并阐述差异存在的可能原因。

5 总结与展望

本项目基于AFL，进行了其在ARM平台上的静态插桩功能的扩展，通过新开发的静态插桩框架，对二进制文件中的使用IDA API搜索到的基本块，插入AFL的代码，并重新打包成ELF文件，使得其能够在无源码的条件下，对ARM平台的ELF程序进行Fuzzing。

对于AFL而言还有一大痛点就是其生成输入的方式，如今AFL生成输入的方式是通过随机找到能探索出程序新路径的输入，然后对该输入做随机变换。这需要大量时间去随机所有的跳转条件，以求得能够进入不同分支的输入。而近期的Angora: Efficient Fuzzing by Principled Search[16]一文中提到了为AFL添加污点分析的办法，来对所有跳转条件做字节级别的污点分析，以求高效找出能进入不同分支的输入。但该方法也仅支持开源程序，如果未来能够把污点分析也结合到本项目的成果里，那也将大大提高对闭源程序的Fuzzing效率。

参考文献

[1] IC Insights. "MCU Market on Migration Path to 32-bit and ARM-based Devices: 32-bit tops in sales; 16-bit leads in unit shipments"[EB/OL]. 25 April 2013. Retrieved 1 July 2014. http://www.icinsights.com/news/bulletins/MCU-Market-On-Migration-Path-To-32bit-And-ARMbased-Devices/

[2] lcamtuf. American fuzzy lop [EB/OL]. http://lcamtuf.coredump.cx/afl/

[3] Brendan Dolan-Gavitt. Fuzzing with AFL is an art [EB/OL]. July 21, 2016. http://moyix.blogspot.com/2016/07/Fuzzing-with-afl-is-an-art.html

[4] lcamtuf. Technical "whitepaper" for afl-fuzz [EB/OL]. http://lcamtuf.coredump.cx/afl/technical details.txt

[5] lcamtuf. Binary Fuzzing strategies: what works, what doesn't [EB/OL]. August 08, 2014. https://lcamtuf.blogspot.jp/2014/08/binary-fuzzing-strategies-what-works.html

[6] Cadar C, Dunbar D, Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs[C] Usenix Conference on Operating Systems Design and Implementation. USENIX Association, 2009:209-224.

[7] Sang K C, Avgerinos T, Rebert A, et al. Unleashing Mayhem on Binary Code[J]. 2012, 19:380-394.

[8] Cadar C, Sen K. Symbolic execution for software testing: three decades later[M]. ACM, 2013.

[9] QEMU document [EB/OL]. https://qemu.weilnetz.de/doc/qemu-doc.html

[10] Torczon L, Cooper K. Engineering A Compiler[M]. Morgan Kaufmann Publishers Inc. 2007.

[11] Hex-rays. IDA Python document [EB/OL]. https://www.hex-rays.com/products/ida/support/idapython_docs/

[12] Cea-sec. Miasm [CP]. https://github.com/cea-sec/miasm

[13] programa-stic. Barf [CP]. https://github.com/programa-stic/barf-project

[14] Radare. Radare2 [CP]. https://github.com/radare/radare2

[15] Cea-sec. Sibyl [CP]. https://github.com/cea-sec/Sibyl

[16] Chen P, Chen H. Angora: Efficient Fuzzing by Principled Search[J]. 2018.

附　录

作者简历

姓名：许晟杰　性别：男　民族：汉　出生年月：1995-12-28

籍贯：福建省南平市

2011.09-2014.06　福建省南平第一中学

2014.09-至今　浙江大学攻读学士学位

获奖情况：

2016.4 浙大 ACM 校赛　二等奖

2017.4 随队(AAA) 参加 BCTF　第二名

2017.5 中控杯机器人比赛　三等奖

2017.6 随队(AAA) 参加 TCTF RSCTF　第一名

2017.8 随队(A*0*E) 参加 DEFCON Final　第三名

2018.5 随队(A*0*E) 参加 PlaidCTF　第一名

2018.5 随队(AAA) 参加 TCTF RSCTF　第三名

本科生毕业论文（设计）任务书

一、题目：基于 American Fuzzy Loop 的静态插桩功能扩展的实现

二、指导教师对毕业论文（设计）的进度安排及任务要求：

1. 通过对已有的AFL工具进行调研，了解目前技术实现机制并以之为基础，以进行后续的改进。

2. 阅读国内外的相关文献，通过文献中的内容寻求改进AFL过程中可能用到的方法。

3. 实现对 AFL 工具的改进，使得其能够在 ARM 架构上对 ELF 程序进行 Fuzzing 工作。

起讫日期 20　年　月　日 至 20　年　月　日

指导教师（签名）＿＿＿＿＿＿职称＿＿＿＿＿＿

三、系或研究所审核意见:

负责人（签名）＿＿＿＿＿＿＿＿＿＿

年　月　日

毕 业 论 文（设计） 考 核

一、指导教师对毕业论文（设计）的评语：

学生设计并实现了一个基于 AFL 的软件安全模糊测试工具，将 AFL 从传统 x86 架构移植到目前流行的 ARM 架构上，并提出和实现了一种针对二进制文件的代码插桩方法，有效提升了 AFL 的执行效率。实验结果表明，该系统能够快速有效对 ARM 架构的二进制可执行程序进行模糊测试，从而为物联网设备软件安全测试与评估提供了一个有效的手段。该选题源自物联网软件安全方向的科研课题，具有很好的创新性和实用性。

指导教师(签名）

年 月 日

二、答辩小组对毕业论文（设计）的答辩评语及总评成绩：

成绩比例	文献综述占（10%）	开题报告占（15%）	外文翻译占（5%）	毕业论文（设计）质量及答辩占（70%）	总评成绩
分值					

答辩小组负责人（签名）

年 月 日

第二部分

开题报告和中期报告