1、Abstract

本文主要是对错误处理代码进行模糊测试，现在的模糊测试技术主要是基于生成和变异的方法来生成种子，尽可能的覆盖代码执行路径，从而发现错误。但是在测试错误处理代码方面非常有限，因为一些错误处理代码只能由偶发错误(例如内存不足和网络连接故障)触发，而不能由特定的输入触发。

所以在本文中作者提出了一个名为FIFUZZ的模糊测试框架，有效的测试错误处理代码。FIFUZZ的核心是上下文相关的软件故障注入(software fault injection，SFI)，该方法可以有效地覆盖错误处理代码，并且可以查找隐藏在具有复杂上下文关系的错误处理代码中的深层错误。

最后FIFUZZ与上下文无关的SFI和现有的模糊测试工具(AFL，AFLFast，AFLSmart和FairFuzz)进行了比较, FIFUZZ发现了这些工具遗漏的许多bug。

2、Background & Introduction

2.1 错误处理代码

错误处理代码是指由于特殊的执行条件，例如用户的无效输入，内存不足和网络连接故障，程序在运行时遇到的特殊情况。我们将这些异常情况称为错误，用于处理错误的代码称为错误处理代码。(例如try / catch中的catch内容)

另外错误可以分为两类：分别是与输入有关的错误和偶发性的错误。与输入相关的错误是由无效输入引起的，例如异常的命令和错误的数据，这类错误可以通过提供特定的输入来触发，目前的模糊测试技术主要也是触发此类错误。

偶发性的错误是由偶尔发生的异常事件引起的错误，例如内存不足或网络连接失败，此类错误与执行环境和系统资源(例如内存和网络连接)的状态有关，但与输入无关，因此现有的模糊测试技术很难触发该错误。

2.2 错误处理代码的特点

> 错误处理代码很重要，但它本身很容易出错
> 错误处理代码很难正确实现，因为它通常涉及特殊和复杂的语义
> 错误处理代码也难以测试，因为此类代码很少执行
> 错误处理代码引起的关注也比较少
> 由于这些特点，错误处理代码中可能会存在许多bug，并且这些bug可能会有比较大的危害。另外通过作者的调研发现，许多CVE漏洞都是由错误处理代码引起的(例如CVE-2019-7846，CVE-2019-2240，CVE-2019-1750和CVE-2019-1785)。

2.3 软件故障注入(software fault injection, SFI)技术

软件故障注入是一种运行时的测试技术，SFI故意将故障或错误注入到被测试程序的代码中，然后执行该程序来测试在执行过程中是否可以正确处理注入的故障或错误。具体而言，将错误注入到能够触发错误处理代码的sites中，我们将每个此类site称为一个error site。许多现有的基于SFI的方法在测试错误处理代码方面都取得了不错的效果。 void fun() { z = malloc(...); // error site if(!z) { // 错误处理代码 ... } }

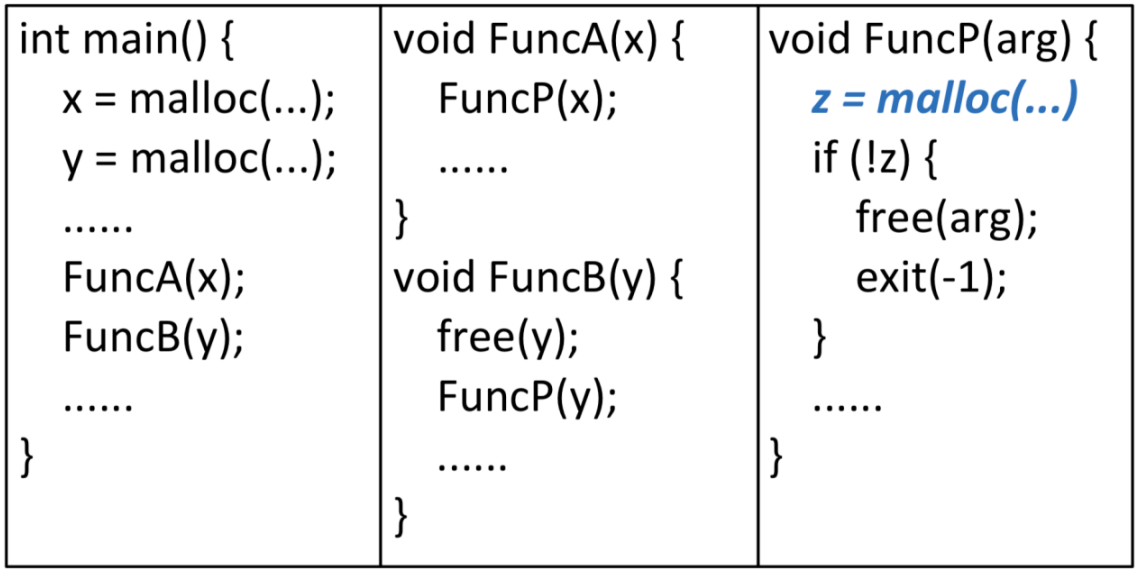
2.4 已有技术检测错误处理代码

> 静态分析（缺少运行时信息，经常引入许多误报
> 输入驱动的模糊测试（因为某些代码只能由非输入偶然错误（例如内存不足和网络连接故障）触发，不能有效地测试错误处理代码
> 现有的基于SFI的测试方法（效果很好，但是他们仅执行上下文无关的故障注入，使测试无法深入的进行

下图解释了现有基于SFI测试方法的局限性。

在主函数中，分配了一个x和y，然后调用了函数FuncA和FuncB，FuncA和FuncB都调用FuncP，但是FuncB在调用FuncP之前释放了传递过来的参数。在FuncP中，通过调用malloc分配对象z，如果malloc调用失败，则释放传递过来的参数，并调用exit退出程序。

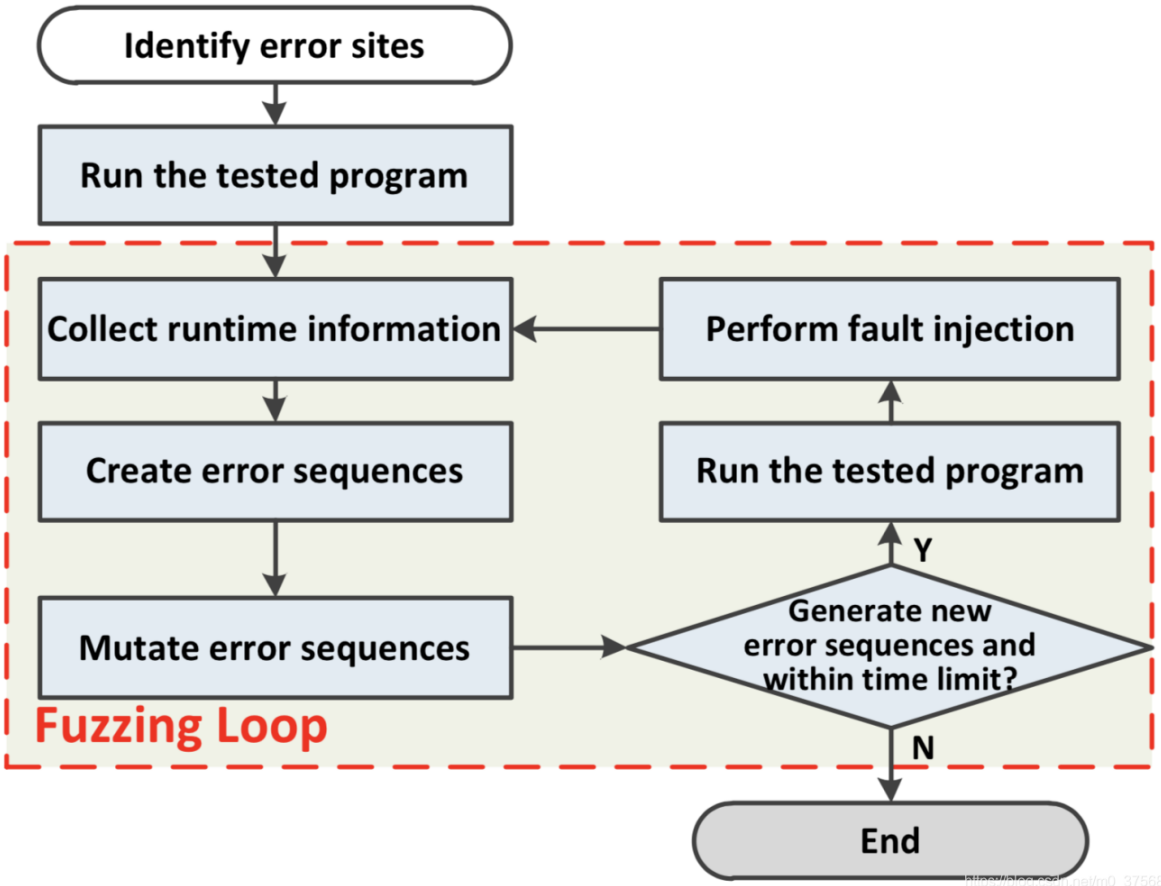
如果使用上下文无关的SFI技术，将故障注入到FuncP中的malloc中，使该函数始终无法调用成功。则在执行FuncA时，程序将始终退出，而不会发现任何错误。



如果使用上下文相关的SFI技术，并且仅在FuncB调用FuncP时才将故障注入到FuncP中的malloc中，则可以在运行时触发对象的双重释放错误。

在本文中，为了有效地检测错误处理代码中的bug，设计了一种基于上下文敏感的SFI的模糊测试方法。该方法将执行上下文纳入考虑范围，以有效地指导SFI以最大程度地发现bug。它包括六个步骤：1）使用静态分析技术识别可能存在错误的位置 2）收集执行error site的上下文调用和代码覆盖率的运行时信息 3）根据运行时的信息创建已执行error site的错误序列 4）对创建的错误序列进行突变生

成新的错误序列 5）将变异后的错误序列注入到程序中 6）收集运行时信息，创建新的错误序列，然后再次对这些错误序列进行变异，从而构成了模糊循环。



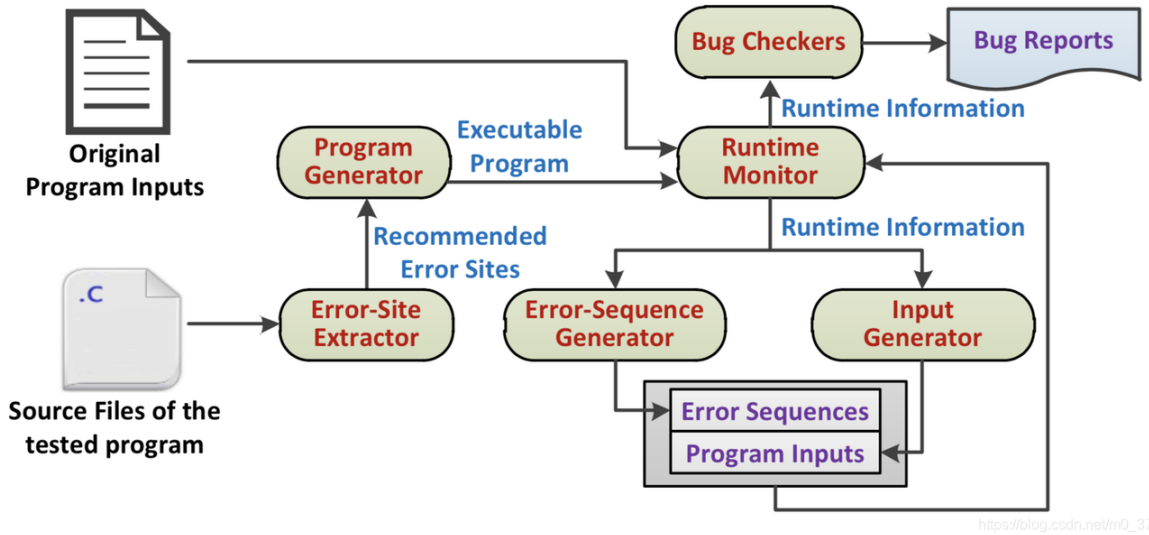
根据作者设计的基于上下文敏感的SFI的模糊测试方法，提出了一个新的模糊框架FIFUZZ。在编译时，为了减少识别error site的人工工作，FIFUZZ对测试程序的源代码进行静态分析，以识别可能的error site。然后，FIFUZZ在运行时测试中使用基于上下文敏感的基于SFI的模糊方法。另外，为了与程序输入的传统模糊测试处理兼容，FIFUZZ通过分析测试程序的运行时信息，将错误序列和程序输入一起突变。

3、FIFUZZ Framework

3.1 FIFUZZ架构

FIFUZZ是使用Clang实现的，对测试程序的LLVM字节码进行代码分析和代码检测。主要包括6部分：

`error site`提取器。它对测试程序的源代码执行静态分析，识别可能的`error site`。
程序生成器。它对程序代码进行代码检测，包括识别出的`error site`，函数调用，函数入口和出口，代码分支等,生成可执行的经过测试的程序。
运行时监视器。它使用生成的输入来运行测试程序，收集测试程序的运行时信息，并根据生成的错误序列进行故障注入。
错误序列生成器。它根据收集的运行时信息创建错误序列，并对错误序列进行突变以生成新的错误序列。
输入生成器。根据收集的运行时信息，它执行传统的模糊处理以变异并生成新的输入。
错误检查器。他们检查收集的运行时信息以检测错误并生成错误报告。



FIFUZZ主要由两阶段组成，分别是Compile-Time Analysis和Runtime Fuzzing

3.2 Compile-Time Analysis

在这个阶段，FIFUZZ主要做两件事情，分别是error site提取和代码检测

error site提取:

error site的提取着重于提取特定的函数，因为作者根据调查发现，大多数error site都与检查函数返回值有关。关于error site的提取主要包括三个步骤：

S1：确定候选的error site。在许多情况下，函数调用会返回空指针或负整数表示调用失败。因此，在以下情况下，我们将函数调用标识为候选error site：1）函数返回了指针或整数； 2）判断返回值是否是NULL或0

```
int [*] getData1() {
    ...
}
getData1() // 候选error site
typedef struct {
    ...
}A;
A getStu() {
    ...
}
if (getStu() == null || getStu() == 0) {

}
```

getStu() // 候选error site S2：选择库函数。在大多数情况下，被测试程序中自定义的函数会执行失败，因为它调用了特定的库函数(例如对malloc进行SFI注入，那么所有使用了malloc的自定义函数都会执行失败)。如果将自定义函数和其调用的库函数都用于SFI，则可能会注入重复的故障。为了避免重复，我们从所有识别的函数调用中选择了那些被调用函数为库函数的函数。

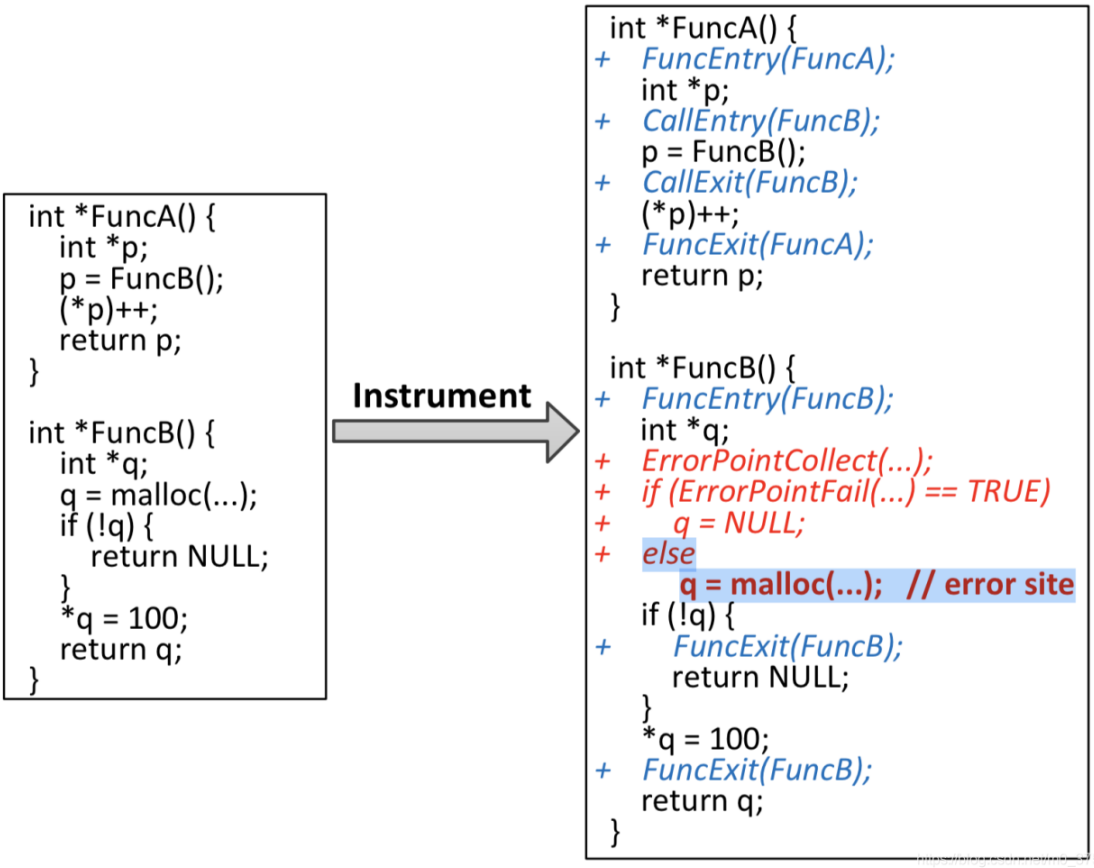
```
void funcA() {
    char *p = (char*)malloc(10 * sizeof(char));
    ...
}
```

// 只对malloc进行SFI，不对funcA进行SFI

S3：进行统计分析。在某些情况下，函数可能会失败并触发错误处理，但是if语句没有对该函数的返回值进行检查。 为了处理这种情况，使用了一种统计方法来提取error site。

代码检测:

代码检测主要用于两个目的：收集有关error site的运行时信息和注入故障。为了收集有关每个error site的运行时调用上下文的信息，程序生成器会在对被测试程序代码中定义的函数进行调用之前和调用之后，函数的入口和出口处对代码进行检测。



此外，为了监视error site的执行并将错误注入其中，程序生成器会在每个error site之前检测代码。在程序执行期间，将收集此error site的运行时调用上下文及其位置以创建错误点。上图显示了C代码中的检测代码示例。注意代码检测实际上是在LLVM字节码上执行的。

3.3 Runtime Fuzzing

Runtime Fuzzing使用传统的模糊处理过程生成的程序输入来执行测试的程序，并使用基于SFI的模糊方法生成的错误序列将故障注入程序中。它还收集有关已执行的错误点，代码分支等的运行时信

息。错误序列生成器根据收集的运行时信息创建错误序列，并进行突变生成新的错误序列。输入生成器执行覆盖率指导的突变以生成新输入。然后，FIFUZZ将这些生成的错误序列和输入组合在一起。

4、Evaluation

4.1 和上下文无关进行比较

Program	FIFUZZ_insensitive			FIFUZZ		
	Useful error sequence	*Alert*	*Bug*	*Useful error sequence*	*Alert*	*Bug*
vim	689	1	1	1,664	58	12
bison	108	3	3	289	11	6
ffmpeg	5	0	0	516	35	12
nasm	7	2	1	78	8	1
catdoc	29	2	2	38	2	3
clamav	29	1	1	325	103	6
cflow	105	1	1	217	1	1
gif2png+libpng	4	0	0	6	0	1
openssl	18	0	0	671	80	8
Total	994	10	9	3,804	298	50

从上图可以看出基于上下文的SFI生成的有用的错误序列更多，测试出来的错误也更多。alert -> 触发警报，同时触发的警报也更多。

4.2 与现有模糊测试工具的比较

选择了四种流行的开源模糊测试工具进行比较，包括AFL，AFLFast，AFLSmart和FairFuzz。同时，为了验证FIFUZZ的通用性，在旧版本2.26（2016年1月发布）的Binutils工具集中中选择了5个常见程序（nm，objdump，size，ar，readelf）作为测试程序。我们使用FIFUZZ和四个模糊测试工具对每个程序进行模糊测试，每个模糊测试的时间限制为24小时。

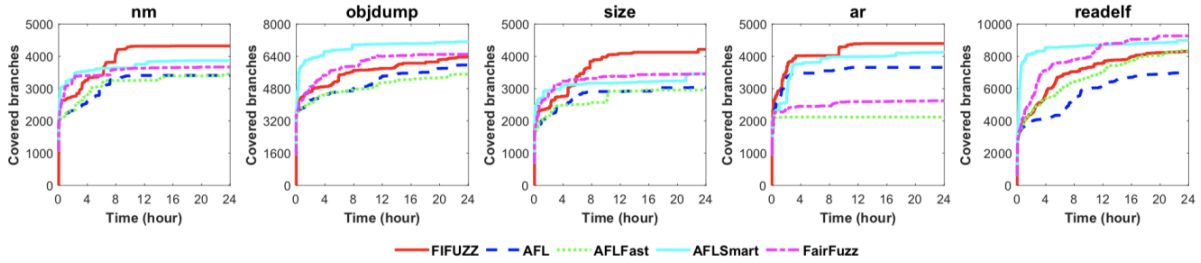


Figure 13: Code coverage of FIFUZZ and the four fuzzing tools.

Program	AFL			AFLFast			AFLSmart			FairFuzz			FIFUZZ		
	Null	*MemErr*	*All*	*Null*	*MemErr*	*All*	*Null*	*MemErr*	*All*	*Null*	*MemErr*	*All*	*Null*	*MemErr*	*All*
nm	0	1	1	0	1	1	0	1	1	0	1	1	4	1	5
objdump	0	1	1	0	1	1	1	1	2	0	1	1	2	1	3
size	0	0	0	0	0	0	0	0	0	0	1	1	2	0	2
ar	0	0	0	0	0	0	0	0	0	0	0	0	4	0	4
readelf	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	0	2	2	0	2	2	1	2	3	0	3	3	12	2	14

Table 9: Results of bug detection for comparison.

图13是模糊测试期间每个测试程序的代码覆盖分支。与AFL和AFLFast相比，FIFUZZ通过覆盖更多的错误处理代码来覆盖所有经过测试的程序中的更多代码分支。与AFLSmart和FairFuzz相比，FIFUZZ在nm，大小和ar上覆盖了更多的代码分支，但是在objdump和readelf中覆盖了更少的代码分支。主要原因是FIFUZZ中程序输入的模糊处理是通过引用AFL来实现的，而AFLSmart和FairFuzz使用一些技术来改进AFU中模糊程序输入的突变和种子选择。出于这个原因，尽管AFLSmart和FairFuzz仍然错过了FIFUZZ覆盖的许多错误处理代码，但它们可以覆盖与输入有关的不经常执行的代码。

表9显示了错误检测的结果。首先，AFLSmart，FairFuzz和FIFUZZ还发现了AFL和AFLFast发现的两个错误。其次，AFLSmart和FairFuzz分别找到了AFL，AFLFast和FIFUZZ遗漏的一个bug。AFLSmart发现的一个额外错误与FairFuzz发现的不同，因为它们以不同的方式改善了程序输入的变异和种子选择。最终，FIFUZZ找到了14个错误，其中12个与错误处理代码有关的错误被AFL，AFLFast，AFLSmart和FairFuzz遗漏了。