

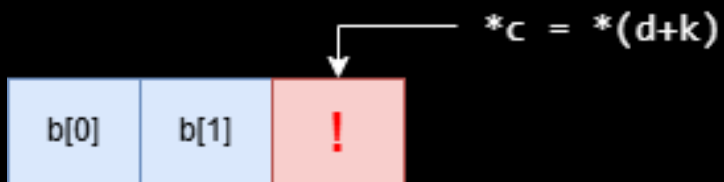
# Fuzzing 相关论文阅读分享

# Part 1 UBfuzz: Finding Bugs in Sanitizer Implementations

# UBfuzz

## 一、内容引出

```
struct a { int x;};  
int main() {  
    struct a b[2];  
    struct a *c=b, *d=b;  
    int k = 0;  
    *c = *b;  
    k = 2;  
    *c = *(d+k);  
    return c->x;  
}
```



### 不开启编译优化

```
henry@henry:~/Documents/test$ clang -O0 -fsanitize=address test.c  
henry@henry:~/Documents/test$ ./a.out  
=====  
==4165==ERROR: AddressSanitizer: global-buffer-overflow on address 0x555555fb95e8  
at pc 0x5555555f4477 bp 0x7fffffffdda0 sp 0x7fffffff570
```

### 开启编译优化

```
henry@henry:~/Documents/test$ clang -O3 -fsanitize=address test.c  
henry@henry:~/Documents/test$ ./a.out  
henry@henry:~/Documents/test$
```

# UBfuzz

## 二、介绍

这篇论文主要是研究 sanitizer 的漏报问题，并提出了一个测试框架的思路，简单来说就是找到存在 UB（未定义的程序行为）的程序，然后再去测试 sanitizer 能否检测到这个问题，如果没有检测到，那么就认为这个 sanitizer 可能存在 bug。

问题引出：

1. 如何生成 UB 程序
2. 编译器优化让 sanitizer 的分析变得复杂化

# UBfuzz

- 解决方案

- 第一个问题

作者提出的解决方案是——**Shadow Statement Insertion**，尽管已经有了可以自动生成程序的工具，如Csmith，但是如果是生成 UB 程序这些工具还不够，文章中采用的思路是，**通过静态分析和动态分析来解析程序的运行时状态**，然后找到一个位置来加入语句，从而引入 UB。拿前面的程序举例，删除掉第七行的 `k=2` 就不存在 UB，反之如果在经过分析后，知道令 `k=2` 会引入溢出，那么就完成了一个 UB 程序的生成

- 第二个问题，


其实并不是说开了 `-O0` 的时候有报告信息，而开了 `-O3` 的时候没有报错，就直接判断 sanitizer 存在问题是不可靠的，原因在于**在编译器优化的过程中，可能会消除掉程序中的一些 UB**，从而导致程序本身相比于 `-O2` 时没有了原来的问题

作者给出的解决方案就是——**crash-site mapping**，将同一个程序编译为两个不同的二进制文件 `b1` (`-O0`) 和 `b2` (`-O2`)，如果 `b1` 存在报错，`b2` 不存在，这时候就通过 debug 的方式，去追踪路径，如果两个二进制程序都能到达崩溃点 (crash-site)，这时候就认为是 sanitizer 产生了漏报问题。

# UBfuzz

## 三、实现方案

```
1 struct a { int x };
2 struct a b[2];
3 struct a *c = b, *d = b;
4 int k = 0;
5 int main() {
    LOG_BufRange(&b[0], sizeof(b)); //-->1
6     *c = *b;
    k = 2; //-->3
    LOG_BufAccess(d+k); //-->2
7     *c = *(d+k);
8     return c->x;
9 }
```

- 
1. 通过静态分析引入分析语句 (1,2)
  2. 动态执行收集 buffer 的内存信息
  3. 根据给定目标生成对应的 UB 语句

**Table 1.** UB conditions and shadow statements. The first three columns describe the conditions for certain code constructs to not have the target UB. The fourth column demonstrates the location where our shadow statements will be inserted. The fifth column presents the effect of each shadow statement. The last column lists the instantiation of each shadow statement in our implementation. Here,  $x, \hat{x}, y, \hat{y}, \hat{c}$  are  $(n + 1)$ -bit integers;  $p, q$  are pointers;  $a$  is an array with capacity  $\text{ARRAYSIZE}(a)$ ;  $lhs \xrightarrow{val} rhs$  represents the value of  $lhs$  is  $rhs$ .

UB	Code Constrcut	Sufficient condition for not having the UB	Shadow Statement $\Delta(\cdot)$	Effect of $\Delta(\cdot)$	Instantiation
Buf. Overflow (Array)	$a[x]$	$0 \leq x < \text{ARRAYSIZE}(a)$	$\Delta(x);$ $\text{Stmt}\{a[x]\};$	$x \xrightarrow{val} v$ and $(v < 0 \vee v \geq \text{ARRAYSIZE}(a))$	$\hat{x} = v - x;$ $\text{Stmt}\{a[x + \hat{x}]\};$
Buf. Overflow (Pointer)	$*p$	$p \in \text{BUFFERRANGE}(p)$	$\Delta(p);$ $\text{Stmt}\{*p\};$	$p \xrightarrow{val} q$ and $q \notin \text{BUFFERRANGE}(p)$	$\hat{c} = q - p;$ $\text{Stmt}\{*(p + \hat{c})\};$
Use After Free	$*p$	$\forall \text{free}(q), !\text{alias}(p, q)$	$\Delta(p);$ $\text{Stmt}\{*p\};$	$p \xrightarrow{val} q$ and $q$ is freed	$\text{free}(p);$ $\text{Stmt}\{*p\};$
Use After Scope	$*p$	$\text{SCOPE}(*p) \in \text{SCOPE}(p)$	$\Delta(p);$ $\text{Stmt}\{*p\};$	$p \xrightarrow{val} q$ and $\text{SCOPE}(*q)$ out of $\text{SCOPE}(p)$	$p = q;$ $\text{Stmt}\{*p\};$
Null Ptr. Deref.	$*p$	$p \neq \text{NULL}$	$\Delta(p);$ $\text{Stmt}\{*p\};$	$p \xrightarrow{val} \text{NULL}$	$p = 0;$ $\text{Stmt}\{*p\};$
Integer Overflow	$x \text{ op } y$	$x \text{ op } y \in [-2^n, 2^n - 1]$	$\Delta(x, y);$ $\text{Stmt}\{x \text{ op } y\};$	$x \xrightarrow{val} v_0, y \xrightarrow{val} v_1$ and $v_0 \text{ op } v_1 \notin [-2^n, 2^n - 1]$	$\hat{x} = v_0 - x, \hat{y} = v_1 - y;$ $\text{Stmt}\{(x + \hat{x}) \text{ op } (y + \hat{y})\}$
Shift Overflow	$x \ll y$ or $x \gg y$	$0 \leq y < n$	$\Delta(y);$ $\text{Stmt}\{x \ll y\};$	$y \xrightarrow{val} v$ and $v < 0 \vee v \geq n$	$\hat{y} = v - y;$ $\text{Stmt}\{x \ll (y + \hat{y})\};$
Divide by Zero	$x/y$ or $x\%y$	$y \neq 0$	$\Delta(y);$ $\text{Stmt}\{x/y\};$	$y \xrightarrow{val} 0$	$\hat{y} = -y;$ $\text{Stmt}\{(x/y + \hat{y})\};$
Use of Uninit. Memory	$\text{if}(x)$ or $\text{while}(x)$	$x$ is uninitialized	$\Delta(x);$ $\text{Stmt}\{x\};$	$x \xrightarrow{val} \text{uninit. memory}$	$\text{int } \hat{x};$ $\text{Stmt}\{x + \hat{x}\};$

First problem solved!

# UBfuzz

活跃区域是指程序执行过程中，某个变量或表达式在某个时刻具有有效值的范围。

## 三、UB Program Generator

### Algorithm 1: UB program generation

```
1 procedure Generator(Program  $\mathcal{P}$ , Input  $\mathcal{I}$ , UBType  $\mathcal{U}$ ):  
    // find all matched expr to a given UB  
2     $E \leftarrow \text{GetMatchedExpr}(\mathcal{P}, \mathcal{U})$   
3     $\widehat{prof} \leftarrow \text{Profile}(\mathcal{P}, \mathcal{I}, \mathcal{U}, E)$     // profiling  
4     $P_{UB} \leftarrow []$   
5    foreach  $expr \in E$  do  
        // synthesize a shadow statement  
6         $\Delta(expr) \leftarrow \text{SynShadowStmt}(expr, \widehat{prof}, \mathcal{U})$   
        // insert the shadow statement  
7         $\mathcal{P}' \leftarrow \text{Insert}(\mathcal{P}, \Delta(expr))$   
        // append the new UB program  
8         $P_{UB}.append(\mathcal{P}')$   
9    return  $P_{UB}$ 
```

### 1. GetMatchedExpr( $\cdot$ )

负责生成  $E$  集合，每个  $E$  中的元素包括对应的语句以及其位置Program

### 2. Profiling — Profile( $\cdot$ )

- 所有在表达式中的值被记录
- 所有堆栈的释放和分配地址信息

这里定义了几个查询语句：

- $Q_{liv}(\widehat{prof}, e)$ : 如果  $e$  位于活跃区域，则返回true，否则返回false。实则就是来检查  $e$  在  $\widehat{prof}$  中是否有值。
- $Q_{val}(\widehat{prof}, e)$ : 返回  $e$  的值
- $Q_{mem}(\widehat{prof}, e)$ : 如果  $e$  是一个指针或者一个数组，则返回  $e$  所指向的内存范围，若已被释放，返回 false
- $Q_{scp}(\widehat{prof}, e)$ : 返回表达式  $e$  的作用域 (scope)，即  $e$  被定义和使用的上下文范围



# UBfuzz

## 三、UB Program Generator

### Algorithm 1: UB program generation

```
1 procedure Generator(Program  $\mathcal{P}$ , Input  $\mathcal{I}$ , UBType  $\mathcal{U}$ ):  
    // find all matched expr to a given UB  
2     $E \leftarrow \text{GetMatchedExpr}(\mathcal{P}, \mathcal{U})$   
3     $\widehat{prof} \leftarrow \text{Profile}(\mathcal{P}, \mathcal{I}, \mathcal{U}, E)$     // profiling  
4     $P_{UB} \leftarrow []$   
5    foreach  $expr \in E$  do  
        // synthesize a shadow statement  
6         $\Delta(expr) \leftarrow \text{SynShadowStmt}(expr, \widehat{prof}, \mathcal{U})$   
        // insert the shadow statement  
7         $\mathcal{P}' \leftarrow \text{Insert}(\mathcal{P}, \Delta(expr))$   
        // append the new UB program  
8         $P_{UB}.append(\mathcal{P}')$   
9    return  $P_{UB}$ 
```

### 3. SynShadowStmt( $\cdot$ ) & Insert( $\cdot$ )

这一部分对前面 Shadow Statement 的生成和插入基于上面的几个查询语句又做了详细说明，具体针对每个 UB 的做法如下：

- *Bufferoverflow(array)*: 引入辅助变量  $\hat{x}$ , 使得原语句变为  $a[x + \hat{x}]$ ,  $\hat{x} = v - x$ , 其中  $x$  来自  $Q_{val}(\widehat{prof}, x)$ ,  $v$  通过计算  $Q_{mem}(\widehat{prof}, v)$  得到
- *Bufferoverflow(pointer)*: 对于指针也是一样的, 跟数组同理
- *Useafterfree*: 不需要使用查询语句, 在 use 前 free 掉就可以了
- *Useafterscope*:  $\Delta(expr)$  为  $p = q$ , 保证  $Q_{scp}(\widehat{prof}, *q)$  的作用范围不在  $Q_{scp}(\widehat{prof}, p)$  中
- *Nullpointerdereference*: 让指针为 0 即可
- *Integeroverflow*: 引入辅助变量  $\hat{x}, \hat{y}$ , 其中  $\hat{x} = v_0 - x$ ,  $\hat{y} = v_1 - y$ ,  $x$  和  $y$  的值直接通过  $Q_{val}(\widehat{prof}, x)$ ,  $Q_{val}(\widehat{prof}, y)$  来获得, 对于  $v_0, v_1$  采用蒙特卡罗算法去在区间内抽样, 使得能够溢出
- *Shiftoverflow*: 同理, 引入辅助变量, 让移位操作溢出
- *Dividebyzero*: 引入辅助变量让分母为 0 即可
- *Useofuninitializedmemory*: 直接引入辅助变量  $\hat{x}$  即可

# UBfuzz

## 四、Crash-site Mapping as the Test Oracle

### Algorithm 2: Crash-Site Mapping

```
1 procedure IsBug(Binary  $b_c$ , Binary  $b_n$ ):  
2    $S_c \leftarrow \text{GetExecutedSites}(b_c)$   
3    $S_n \leftarrow \text{GetExecutedSites}(b_n)$   
4   if  $S_c[-1] \in S_n$  then  
5     return True  
6   else  
7     return False  
8 procedure GetExecutedSites(Binary  $b$ ):  
9    $S \leftarrow []$   
10   $\text{debugger.Init}(b)$   
11  while  $\text{debugger.IsAlive}()$  do  
12     $l \leftarrow \text{debugger.curr\_line}$   
13     $o \leftarrow \text{debugger.curr\_offset}$   
14     $S.\text{append}((l, o))$   
15     $\text{debugger.NextInstruction}()$   
16  return  $S$ 
```

*crash site*: A binary  $b_i$  is compiled from program  $\mathcal{P}$  and running  $b_i$  results in a crash. We denote the last executed instruction as *inst*. If *inst* corresponds to the line  $l$  and offset  $o$  in  $\mathcal{P}$ , then the crash site of  $b_i$  is  $(l, o)$ .

$b_c$  是未经优化编译的程序

$b_n$  是经过优化编译的程序

但是这时候为了确定是编译器优化导致消除了 UB，还是 ASAN 出现漏报，所以就需要进行判断

算法所采用的思路就是记录程序运行时的状态信息（ $\text{curr\_line}$  和  $\text{curr\_offset}$ ），如果  $b_n$  中执行了  $b_c$  最后的崩溃状态，此时返回 True，即认为是 ASAN 的漏报问题。

second problem solved!

# UBfuzz

## 五、Detail

UBFuzz 总共包含 2000 行 c++ 代码, 4400 行 python 代码

具体实现如下:

- Clang 的 `libtools` 实现表达式匹配
- 程序插桩实现 shadow statement 插入分析
- LLDB 跟踪记录 Crash-Site Mapping 中所需的信息, 并使用其支持的 python API 来完成整个自动化分析过程

### Compilers and sanitizers

选择使用 GCC 和 llvm, 以及 ASAN, UBSAN 和 MSan。

### Seed programs

选择使用 Csmith 来完成

# UBfuzz

## 六、Samples

```
1 int g, *ptr = &g;
2 int **p_ptr = &ptr;
3 int main() {
4     int buf[3]={1,2,3};
5     *ptr = 1;
6     *p_ptr = &buf[3];
7     *ptr = 0xfff;
8 }
```

(a) GCC ASan at -O1 missed the buffer overflow access \*ptr at line 7. [7]

```
1 volatile int a[5];
2 void b(int x) {
3     if(x)
4         a[5] = 7;
5 }
6 int main(){ b(1); }
```

(d) LLVM's ASan missed the buffer overflow at line 4. [19]

```
1 int a, c;
2 short b;
3 long d;
4 int main() {
5     a = (short)(d == c |
6             b > 9) / 0;
7     return a;
8 }
```

(b) GCC's UBSan at all levels missed the division-by-zero at line 5. [9]

```
1 int main() {
2     int *a = 0;
3     int b[3]={1, 1, 1};
4     ++b[2];
5     ++(*a);
6 }
```

(e) LLVM's UBSan missed the null pointer dereference at line 5. [20]

```
1 void b() {          9     for(;a<=5;++a){
2     int c[1];        10         int f[1]={};
3     c;               11         e = f;
4 }                   12         a||(b(), 1);
5 int main() {        13     }
6     int d[1]={1};    14     return *e;
7     int *e = d;      15 }
8     a = 0;
```

(c) GCC's ASan missed the use after scope at line 14, where the pointer e points to an inner scope variable f defined at line 10. [8]

```
1 int main() {
2     unsigned char a;
3     if (a-1)
4         __builtin_printf("boom!\n");
5     return 1;
6 }
```

(f) LLVM's MSan missed the use of uninitialized memory at line 3. [21]

Figure 12. Sample UB programs that trigger sanitizer FN bugs.

# Part 2 Fuzz4All: Universal Fuzzing with Large Language Models

# Fuzz4All

## 一、介绍

这篇论文针对的是对以编程语言或形式语言作为输入的系统（编译器，运行引擎，约束求解器）进行有效 fuzz 而提出的一种模糊测试模型，之所以针对这类系统进行模糊测试，是因为它们的 bug 会严重影响一些下游产品的安全性。

### 背景:

传统的 fuzzer 可以被归类为基于生成的和基于变异的（混合模型），但这种传统的模糊测试技巧面临以下限制和挑战：

- 目标系统和语言的紧耦合  
比如说针对某种语言的模糊测试器，放到另外一种语言下就不能再使用了
- 缺乏对新特性的支持  
比如说传统模糊测试器只针对某种语言的某一个版本，而语言也是慢慢更新的，但这些模糊测试器并没有将这些更新的新特性加入到模糊测试器的考虑范围之内
- 受限的生成能力  
基于生成的模糊器其通常非常依赖输入语法和一些语义规则来确保输入的有效性，但这在一定程度上会回避一些难以建模的语言特征，这使得只覆盖到语言的部分特性，基于变异的模糊器也是如此

# Fuzz4All

## 一、介绍

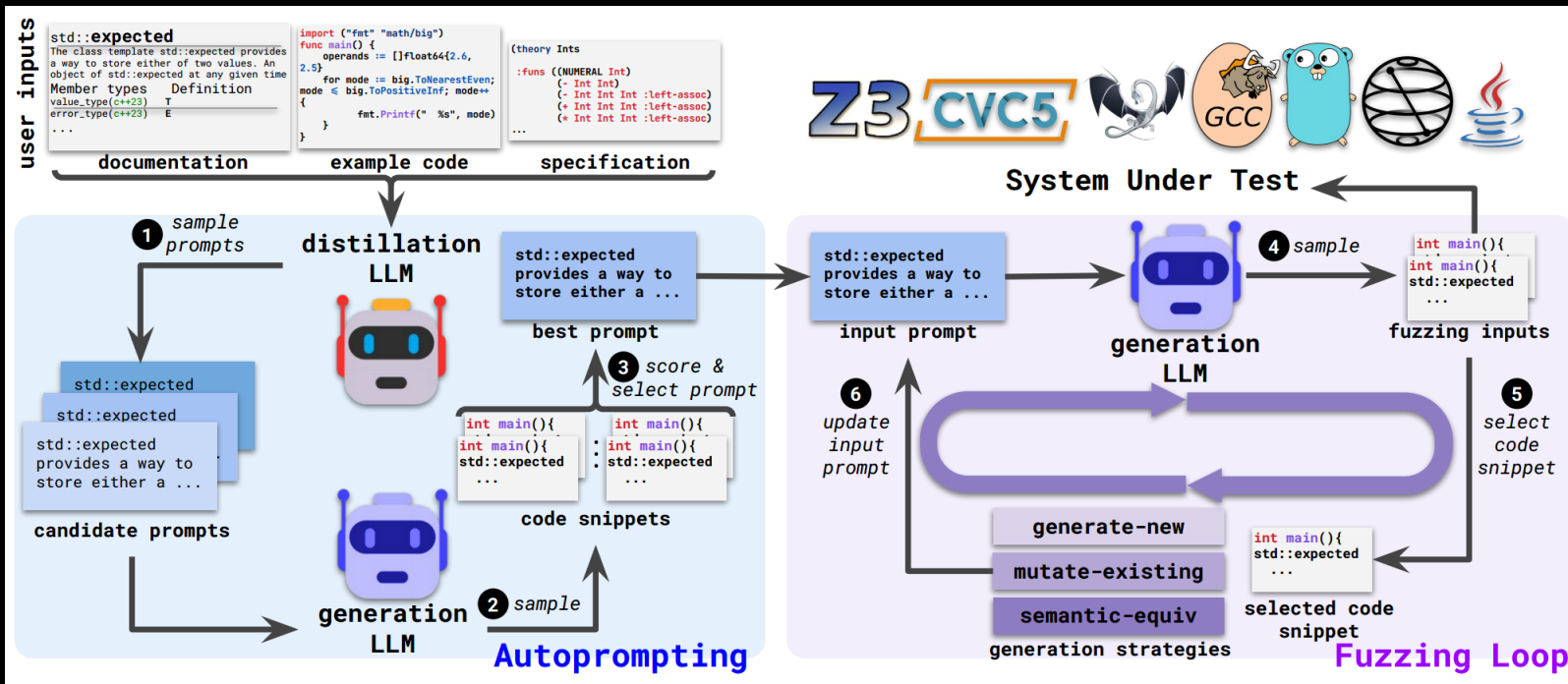
### 贡献:

这篇论文的工作：提出了一个可以针对不同语言输入（C, C++, GO）和这些语言不同的特征都能够有效进行模糊测试的**基于 LLM 的模糊器**。这个可以很好解决上面提出的三个问题，同时这篇文章做出的贡献在于：

- 利用 LLM 生成广泛有意义的输入进行测试
- 通过自动将用户输入提取为有效生成 SUT(system under test) 输入的 prompt
- 迭代修改 prompt，连续生成新的模糊输入
- 在真实世界的有效性

# Fuzz4All

## 二、FUZZ4ALL APPROACH



### Autoprompting

Fuzz4All 通过使用 DLLM 来生成 candidate prompts

每个 candidate prompt 被传递给 GLLM 以生成代码片段

然后从生成的代码片段中进行评分，并选出 best prompt

### Fuzzing Loop

上一步骤中的 best prompt 会作为 Fuzzing Loop 的 input prompt

这一阶段不断通过 GLLM 生成模糊测试的有效输入

同时为了避免输出相同的输入示例，Fuzz4All 每次迭代的过程中都会更新 input prompt 来保证在下一个循环阶段，生成不一样的输入示例。



# Fuzz4All

## 三、Autoprompting

从前面的图可以看到，distillation LLM 接受技术文档，示例代码，包括代码规范作为输入，不像传统的 fuzzer 那样要求特定的输入格式，同时由于用户输入的一些信息可能是冗余的或者不相关的，所以 autoprompting 的目标正是剔除这些因素，生成有效的基于 LLM 模糊测试的 input prompt。

### Algorithm 1: Autoprompting for fuzzing

```
1 Function Autoprompting:
   Input :userInput, numSamples
   Output:inputPrompt
2   greedyPrompt  $\leftarrow \mathcal{M}_{\mathcal{D}}$  (userInput, APIInstruction, temp=0)
3   candidatePrompts  $\leftarrow$  [ greedyPrompt ]
4   while |candidatePrompts| < numSamples do
5     prompt  $\leftarrow \mathcal{M}_{\mathcal{D}}$  (userInput, APIInstruction, temp=1)
6     candidatePrompts  $\leftarrow$  candidatePrompts + [ prompt ]
7   inputPrompt  $\leftarrow \arg \max_{p \in \text{candidatePrompts}} \text{Scoring}(\mathcal{M}_{\mathcal{G}}(p), \text{SUT})$ 
8   return inputPrompt
```

左图为 Autoprompting 实现的主要的一个算法逻辑，算法的实现思想也很简单，最终返回一个 input prompt

这里的 Fuzz4All 首先使用温度为0的贪婪采样生成一个 candidate prompt，该算法通过低温先采样，得到具有高置信度的可信解，然后，算法继续在更高的温度下进行采样，以获得更多样化的 prompt

在经过 DLLM 的处理后，会生成多个 candidate prompts，为了评估这些 prompt 的有效性和表现，这里使用 GLLM 生成对应这些 prompt 的代码片段，然后会对这些代码片段进行评分，从而选出最优的 prompt 用于下一阶段的训练

评分的依据可以是覆盖率，发现的bug数，或者生成的输入示例的复杂性，最终计算出来的最高分即为 fuzzing loop 所需要的 input prompt

# Fuzz4All

## 四、Fuzzing Loop

Fuzzing loop 的核心思想是通过从先前的迭代中选择一个示例模糊测试输入并指定生成策略，不断地增加原始 input prompt。在每次新的模糊循环迭代之前，Fuzz4All 将一个示例和生成策略附加到输入提示符中，使生成 LLM 能够不断地创建新的模糊输入。

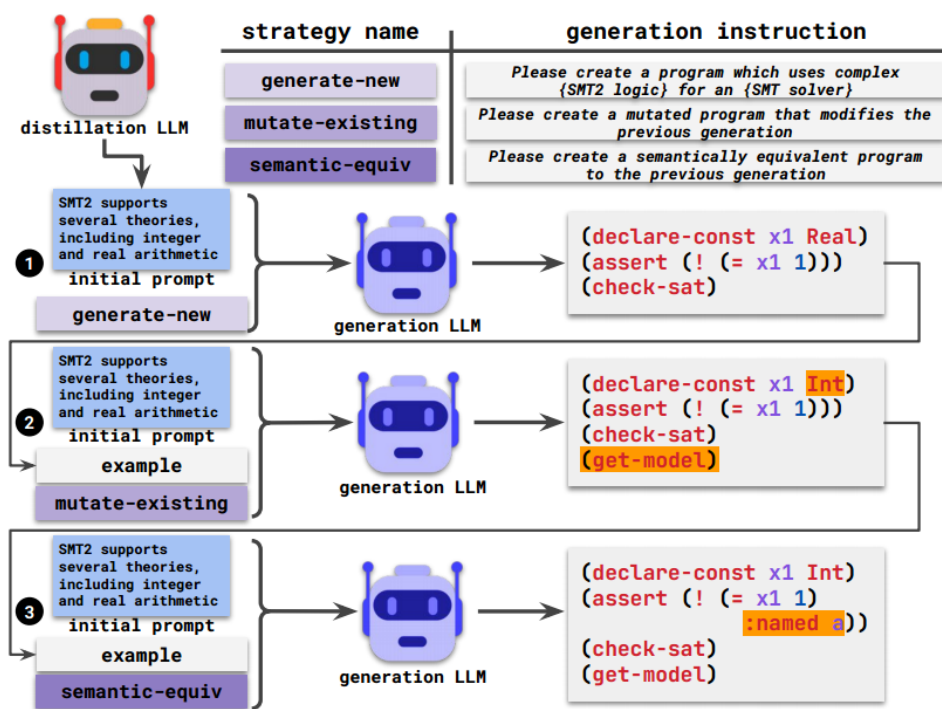


Figure 3: Fuzzing strategies and example of fuzzing loop.

可以从左图理解整个 fuzzing loop 所做的事，其实归纳为三个核心点就是：

- **generate-new** 通过 initial prompt 和 generate-new 指令，让 GLLM 生成一个代码片段示例，该示例作为下一个步骤的 example
- **mutate-existing** 这一步将上一步产生的 example 进行变异（其它输入都没有变化），然后交给 GLLM 再次生成变异后的代码片段
- **semantic-equiv** 这一步顾名思义产生在前面代码示例的基础上生成另外一种语法上等价代码片段表示，比如在上面的例子中多了个 `:named a`，虽然在语法上是等价的，但在找 bug 时，这种替换却是有效的

# Fuzz4All

## 四、Fuzzing Loop

左图即为 Fuzzing loop 整个过程的伪代码实现， $M_G$  即代表 GLLM，首先初始化一个  $genStrats$  的向量，里面分别定义了前面提到的三种策略，其次第三行生成第一批原始模糊测试示例。

第5行到第9行是 fuzzing loop 的**主要部分**：

- 第6行会从先前产生的所有输入示例中随机选择示例用来测试
- 同时，第7行又从  $genStrats$  的三种策略中随机选择一种作为 GLLM 的下一步输入（第8行）
- 如果用户定义的  $oracle$  识别出意外的行为，例如，崩溃，那么算法将向检测到的错误集合添加一个（第9行）
- 最后，循环的终止条件设置为了耗尽所有 fuzzing budget 时停止

### Algorithm 2: Fuzzing loop

```
1 Function FuzzingLoop:
   Input :inputPrompt, timeBudget
   Output: bugs
2    $genStrats \leftarrow [generate\_new, mutate\_existing,$ 
      $semantic\_equiv]$ 
3    $fuzzingInputs \leftarrow M_G(inputPrompt + generate\_new)$ 
4    $bugs \leftarrow Oracle(fuzzingInputs, SUT)$ 
5   while timeElapsed < timeBudget do
6      $example \leftarrow sample(fuzzingInputs, SUT)$ 
7      $instruction \leftarrow sample(genStrats)$ 
8      $fuzzingInputs \leftarrow M_G(inputPrompt + example +$ 
        $instruction)$ 
9      $bugs \leftarrow bugs + Oracle(fuzzingInputs, SUT)$ 
10  return bugs
```

$Oracle$  这里的功能是根据输入示例，来检查对应的 SUT 的行为以检测错误，这个  $Oracle$  可以由用户定义，比如说我们可以将 segfault 和内部断言失败作为定义  $Oracle$  检测错误的标准

# Fuzz4All

## 五、Implementation

Fuzz4All主要由 Python 实现

与传统的fuzzers(如Csmith (>80K LoC))相比, Fuzz4All 具有非常轻量级的实现

由前面的内容介绍可以知道, Fuzz4All 实际上由两个关键阶段构成即 Autoprompting 和 Fuzzing Loop, 而这两个阶段的核心构成组件为 DLLM 和 GLLM。

Fuzz4All 使用 GPT4 作为 DLLM 来执行 Autoprompting, 对于 Fuzzing Loop, 使用 StarCoder 模型的hug Face实现作为 GLLM。

## 六、Strength

- 在传统 Fuzz 加上了 LLM
- 在路径探索上表现出不错的效果
- 如论文的名字一样 Fuzz4All, 实现了针对多个目标进行模糊测试
- 由于 LLM 的灵活性, Fuzz4All 可以根据当下的一些新特性来产生特定于其他模糊器的输入

# Part 3 Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation

# PathEval

## 一、介绍

主要研究的是大语言模型在生成定向输入测试集时的性能，还有和已有的约束求解器之间的比较。

首先文章提出自动化测试对于程序的可靠性和安全性有着极为重要的作用。这些自动化测试技术包括 fuzzing，符号执行等，但这些都面临着一个相同的问题就是如何提高测试覆盖率。

### 背景:

```
1  int execute(char* s) {  
2      int ret = system(s);  
3      if(ret == 0){  
4          // target line  
5      }  
6  }
```

**Figure 1: An example that symbolic execution tools cannot solve while LLMs can.**

在左边这个例子中，为了执行到 target line，约束求解器很难有效完成这个工作，实际上只要输入一个完整有效的 linux 命令即可，但相反 LLM 却可以通过指定字符串 *s* 的内容，来轻松完成这个工作。

# PathEval

## 一、介绍

主要研究的是大语言模型在生成定向输入测试集时的性能，还有和已有的约束求解器之间的比较。

首先文章提出自动化测试对于程序的可靠性和安全性有着极为重要的作用。这些自动化测试技术包括 fuzzing，符号执行等，但这些都面临着一个相同的问题就是如何提高测试覆盖率。

### 背景:

```
1  int arr(char *s){  
2      int symvar = s[0] - 48;  
3      int ary[] = {1,2,3,4,5};  
4      if (ary[symvar%5] == 5){  
5          // target line  
6      }  
7  }
```

Figure 2: An example that LLMs cannot solve while symbolic execution tools can.

但同时作者又通过另外一个例子来说明在有些情况下，LLM 不能完成的但是约束求解器可以完成。在左图中，需要设置一个 `s[0]` 使得最后能通过条件语句，进入到目标行，这一点约束求解器完全可以做到求解出具体的 `s[0]`。但同样的，LLM 却在完成这项任务时有些乏力。

# PathEval

## 一、介绍

主要研究的是大语言模型在生成定向输入测试集时的性能

首先文章  
执行等，

背景:

```
1  int execute(char* s) {  
2      int ret = system(s);  
3      if(ret == 0){  
4          // target line  
5      }  
6  }
```

Figure 1: An example that symbolic execution tools cannot solve while LLMs can.

```
1  int arr(char *s){  
2      int symvar = s[0] - 48;  
3      int ary[] = {1,2,3,4,5};  
4      if (ary[symvar%5] == 5){  
5          // target line  
6      }  
7  }
```

Figure 2: An example that LLMs cannot solve while symbolic execution tools can.

通过这两个例子其实不难总结出，LLM 和 约束求解器各自的优势和劣势，在需要一些外部知识加持的上下文时（比如第一个例子，要给出一个有效的命令），LLM 是有优势的，但是在面对一些需要精确计算的案例时，约束求解器却展现出了其优势

这时候也就自然引出了一个想法，那么就是如果将这两者结合起来，是否能够让生成的输入集更强大（能够满足更高的覆盖率），事实上经过作者团队的测试，这样确实可以有效提高输入集生成的成功率



# PathEval

## 一、介绍

主要研究的是大语言模型在生成定向输入测试集时的性能，还有和已有的约束求解器之间的比较。

首先文章提出自动化测试对于程序的可靠性和安全性有着极为重要的作用。这些自动化测试技术包括 fuzzing，符号执行等，但这些都面临着一个相同的问题就是如何提高测试覆盖率。

### 贡献:

- **Originality**: 系统研究 LLM 生成测试输入，并且与现有的约束求解器进行比较
- **Findings**: LLM 和约束求解器都有其各自的优点和缺点
- **Perspective**: 基于上一步骤，将两者合并起来，并在生成测试输入上展现出不错的效果
- **Open-Source**: 文章中的所有代码已开源，<https://github.com/CGCL-codes/PathEval>

# PathEval

## 二、Background and Motivation

定向测试输入 (directed test input) 生成的目标是生成基于给定目标的测试用例。在生成定向测试输入一般存在两种方法：

### 1. Constraint-based Approaches

基于约束生成测试输入是一种经典的方法，这种方法又可以划分为符号执行和动态符号执行，顾名思义这些工具是通过求解约束来探索程序中的执行路径。在灰盒模糊测试当中，一般会将那些难以覆盖的分支交给符号执行的这些工具以方便来达到更好的性能。

### 2. LLM-based Approaches

论文指出最近的研究表明LLM也可以用于测试输入的生成，从而让覆盖率提高，但是LLM存在的一个问题就是有时会存在错误的输出。同时LLM和约束求解器之间有效性也没有被深究过（本篇论文的重心）。

# PathEval

## 三、Experimental Setup

1. LLMs 在生成定向测试输入方面的表现如何
2. LLMs 相比如传统的约束求解器，其性能如何
3. 是否可以将 LLMs 融入到这些约束求解器中，以获得更好的性能

这一部分论文主要指出已有的 benchmark 在测试文章中讨论的问题时存在的一些缺陷，目前的数据测试集有 `logic_bomb` 和 `CGC`，但这两个都有各自的缺点，`logic_bomb` 的样本数量太少（53 个），不足以支持实验研究，而 `CGC` 的数据集是针对内存不安全语言（C/C++）生成的，因此其并不能反应真实的开发场景

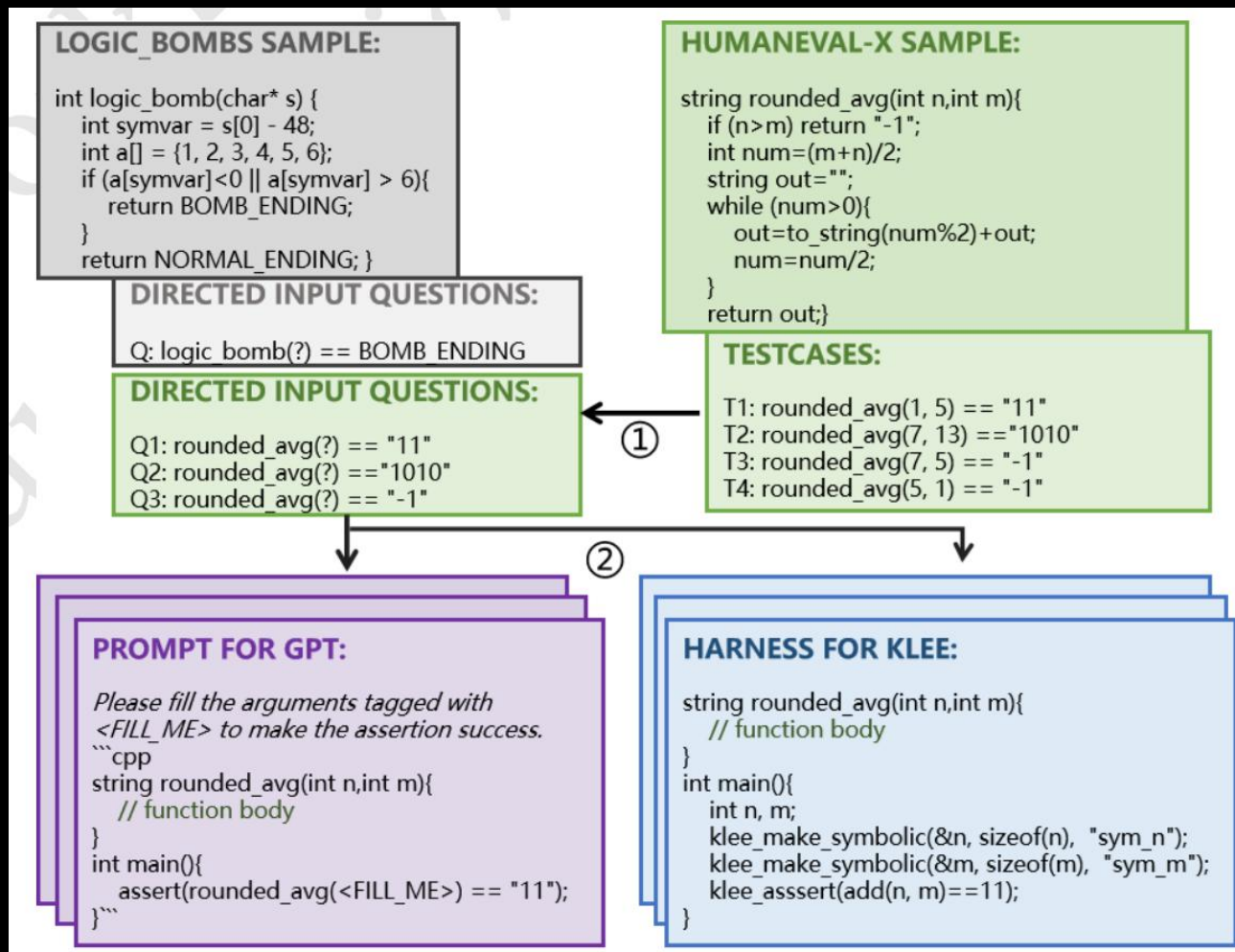
**Table 1: The datasets used in this study. “Lang.” and “Num.” refer to the language and number of the samples in the dataset, respectively.**

Dataset	Desc.	Lang.	Num.
<i>logic_bombs</i>	Dataset for symbolic execution tools.	C	53
<i>PathEval</i>	Reconstructed from <i>HumanEval-X</i> .	C++	741
		Python	747
		Java	744

因此，基于以上问题，论文中所采用的方法就是通过扩展已有的数据集来达成目的，这里选择的是 HumanEval 作为原始数据集，OpenAI 引入的 [HumanEval](#) 是目前最著名的用于评估 LLM 在代码任务上的性能的基准，为了[满足多语言这一特性](#)，使用 HumanEval-X 作为原始数据集，将 HumanEval-X 进行[扩展后的数据集](#)为 [PathEval](#)。

# PathEval

## 三、Experimental Setup



这里所采用的测试思路也是非常简单的，先通过正向输入来得到最终结果（绿色图部分给出了具体过程），如 T1:rounded\_avg(1,5)=="11"，这样就算是一个测试用例，在交由 LLM 进行求解时，则将对应 rounded\_avg 中的参数进行替换，只给出结果，进行逆向求解，并且生成对应 prompt 交给 GPT 进行回答（紫色部分），同时左图也给出了使用 KLEE 进行测试时的方法（蓝色部分）。

# PathEval

## 三、Experimental Setup

在与代码相关工作有关的LLMs中，有如下比较优秀的开源和闭源大模型：

- [GPT3.5](#)：已被用于一些前沿的工作，另外其花费也比 GPT4 少的多
- [StarCoder2](#)：用于代码生成，选择使用 7B（模型所使用的参数）模型，15B在性能上显示出了一定的缺陷
- [CodeLlama](#)：有着比较好的性能
- [CodeQwen](#)：在 code-related tasks 排行榜中表现最好

同时，也有一些常用的符号执行工具：

- [KLEE](#)：支持C\C++的符号执行工具
- [Angr](#)：支持多架构，以及程序分析等，在逆向，CTF，二进制分析中使用广泛
- [CrossHair](#)：支持 python 的符号执行工具
- [EvoSuite](#)：支持 java 的符号执行工具

参考链接

<https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard>

# PathEval

## 四、Result

**Table 2: The performance (i.e., pass rate) of studied LLMs on the two datasets. The “Lang.” column indicates the language of the samples. The “Att.” column indicates the results under single and five attempts, respectively. The red coloring columns indicate the gap between the tool’s performance and that of *GPT-3.5*.**

Dataset	Lang.	Att.	GPT-3.5 (%)	CodeLlama (%)	StarCoder2 (%)	CodeQwen (%)			
logic-bomb	C	1	35.85	13.21	-22.64	9.43	-26.42	28.30	-7.55
		5	43.40	15.09	-28.31	13.21	-30.19	37.74	-5.66
PathEval	C++	1	38.06	30.23	-7.83	12.55	-25.51	18.89	-19.16
		5	58.57	36.30	-22.27	22.94	-35.63	28.34	-30.23
	Python	1	36.81	28.66	-8.15	14.46	-22.36	26.51	-10.31
		5	53.55	34.15	-19.40	19.28	-34.27	38.15	-15.39
	Java	1	31.05	11.91	-19.14	10.89	-20.16	23.92	-7.12
		5	51.21	16.38	-34.83	15.19	-36.02	37.10	-14.11

这张图反映了各个大模型在生成测试输入方面的性能表现

其中 lang 表示的是对应的编程语言  
Att. 代表的是同一个样本所测试的次数

根据左图可以发现，连续测试5次的情况下的结果是远远好于只测试1次的情况（所以这也给了我们一些使用大模型的思路，就是多测试几次，结果会更好一点）。同时也可以通过表格中的数据发现 *GPT-3.5* 的效果是远远好于其他 LLMs 的，展现了 GPT 的强大。

# PathEval

## 四、Result

Table 3: The performances (i.e., pass rate) of constraint-based tools, *GPT-3.5* and *CodeLlama*. “GPT”, “CL”, “CH”, and “ES” represent *GPT-3.5*, *CodeLlama*, *CrossHair* and *EvoSuite*. Both the result of *GPT-3.5* and *CodeLlama* are in five attempts. “-” means the tool does not support the language.

Dataset	Lang.	GPT (%)	CL (%)	Angr (%)	KLEE (%)	CH (%)	ES (%)
<i>logic-bombs</i>	C	43.40	15.09	28.30	24.53	-	-
<i>PathEval</i>	C++	58.57	36.30	-	59.51	-	-
	Python	53.55	34.15	-	-	32.53	-
	Java	51.21	16.38	-	-	-	39.25

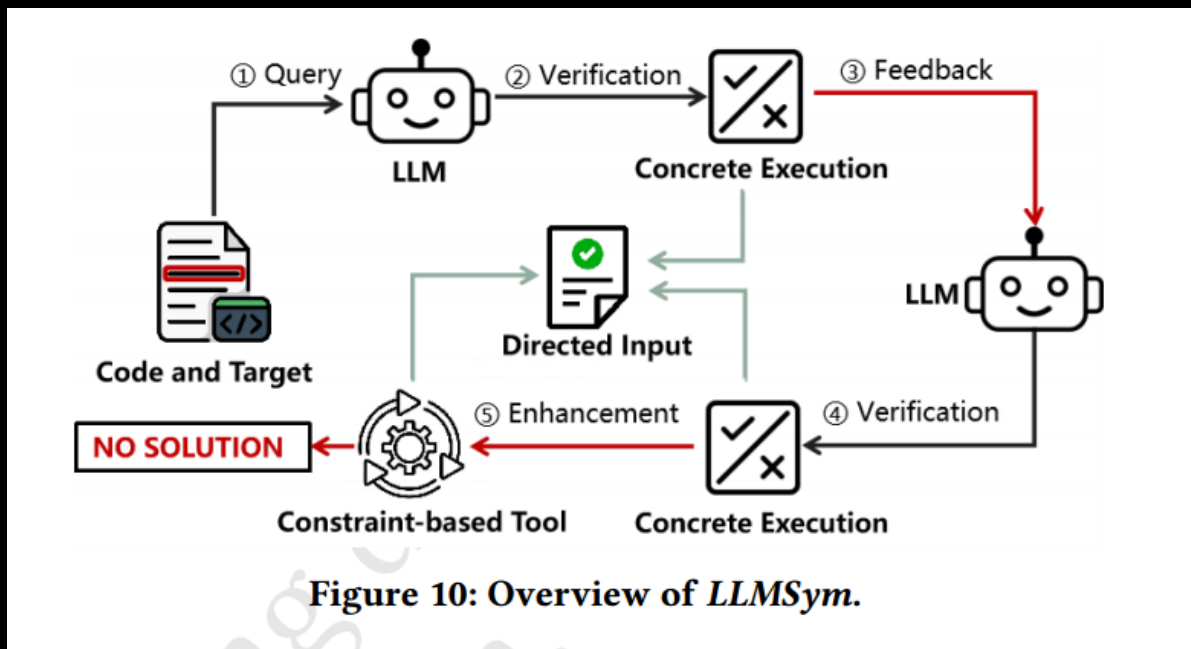
Table 4: The categorized result of *GPT-3.5*, *KLEE* and *Angr* on *logic\_bombs* dataset.

Categeroy	GPT-3.5	KLEE	Angr	Total
buffer_overflow	0	2	2	4
contextual_symbolic_value	2	2	0	5
covert_propagation	6	1	2	7
crypto_functions	0	0	0	2
external_functions	5	0	2	8
floating_point	3	0	2	5
integer_overflow	1	2	2	2
loop	1	0	0	5
parallel_program	2	0	0	5
symbolic_jump	0	1	0	3
symbolic_memory	3	5	4	7
Total	23	13	14	53



# PathEval

## 五、LLMSys



这一部分简单介绍了一下LLM和传统约束求解器结合之后的架构图，从上图能够大致看明白整个原理如何，事实上，对于这一部分内容也论文中只用了半页内容，稍显粗糙。

同时从上面的这个 *LLMSyms* 的架构来看，实际上只是将步骤分开了，LLM 处理两次，然后在最后不能产生有效输入的情况下再交给约束求解器，这实际上并没有发挥 LLM 更大的优势，比如说让 LLM 指导约束求解器（比如在需要某些外部输入的情况下）生成结果，那这样效果肯定是要更好的，上图中实际上性能虽然会提升，但本质上就是让这两个东西合起来各跑一遍。



# Part 4 DeGPT: Optimizing Decompiler Output with LLM

Paper Link: <https://www.ndss-symposium.org/wp-content/uploads/2024-401-paper.pdf>

Code Link: <https://github.com/PeiweiHu/DeGPT>

THANKS