華南師範大學

South China Normal University

本 科 毕 业 论 文

论文题目

不稳定代码静态分析系统的设计与实现

指导老师 王立斌

学生姓名 陈广庆

学　　号 20152100024

院　　系 计算机学院

专　　业 网络工程

毕业时间 2019 年 6 月

摘　要

本文设计实现一种不稳定代码静态分析检查器，该系统以最新 LLVM 框架为开发基础，使用多种高效静态分析算法。

程序未定义行为在 C/C++ 等系统编程语言中广泛存在，然而编译器开发者与编译器使用者对其的认知却有着一定的分歧。导致相关代码被编译器优化后会出现功能差异甚至严重的**安全漏洞**。这种**不稳定代码**现象在近几年间理论上有了初步的研究，但实践上完全未普及开来，甚至依然存在于很多大型系统中。

相比现有的多年未维护的类似实现，本静态检查器基于最新的编译框架、中间代码扫描管理器、可满足性模理论求解器等进行重新设计，并且优化了缓存机制，大大提高了效率，能够静态分析出以 Clang/GCC 作为编译器的系统中的特定几类不稳定代码及相关安全漏洞, 之后也将持续开发，添加更多支持。

关键词：安全漏洞；未定义行为；不稳定代码；静态分析

ABSTRACT

Program undefined behaviors are widespread in system programming languages such as C/C++, However, compiler developers and compiler users have some differences in their understanding of undefined behavior, which causes some code behave to be inconsistent with the original intention and sometimes causes serious security vulnerabilities. This **unstable code** phenomenon has been theoretically studied in recent years, but it has not yet well known in practice, and even exists in many large systems.

We design and implement an unstable code static checker. The system is based on the latest LLVM framework and uses a variety of efficient static analysis algorithms. Compared to existing similar implementations that have not been maintained for many years, this static checker is redesigned based on the latest compilation framework, pass manager and SMT solver. It also optimizes the caching mechanism, which boost the performance. The checker can statically analyze certain types of unstable code and related security vulnerabilities in system compiled with Clang/GCC, and will be developed continuously.

KEY WORDS: security vulnerabilities; undefined behavior; unstable code; static analysis

目 录

摘 要 I
ABSTRACT III
第一章 绪论 1
1.1 引言 1
1.2 研究背景 3
1.3 本文工作与结构安排 3
第二章 基础知识 5
2.1 不稳定代码的定义 5
2.2 静态检测不稳定代码的算法 5
2.2.1 良定义程序假设 6
2.2.2 消除不可达代码 6
2.2.3 简化不必要运算 6
2.3 LLVM 框架 7
2.3.1 LLVM IR 7
2.3.2 LLVM Pass 7
第三章 系统设计 9
3.1 生成中间语言 9
3.2 转换中间语言 9
3.2.1 简化中间语言 9
3.2.2 插入未定义行为条件 10
3.3 分析中间语言 11
3.3.1 检测会被消除的不稳定代码 11
3.3.2 检测会被简化的不稳定代码 11
第四章 系统实现 13
4.1 介入编译过程 13
4.2 中间语言扫描 13
4.2.1 BugOnPass 13
4.2.2 BugFreePass 13
4.2.3 Diagnostic 14
4.3 可满足性模理论求解 14
4.3.1 统一接口 14

4.3.2 简化求解 14
4.3.3 缓存结果 14
第五章 系统测试 15
5.1 样例测试结果 15
5.2 PostgreSQL 测试结果 16
第六章 总结与期望 17
6.1 总结 17
6.2 展望 17
参考文献 19
致 谢 21

第一章　绪论

1.1　引言

很多程序语言在制定标准的时候，往往会申明某些行为是未定义行为，比如 C 语言标准申明空指针解引用为未定义行为。

未定义行为主要有两类，一类是编程错误，比如缓冲区溢出，另一类是不可移植操作，不同硬件在同样的代码下会有微妙的行为差异，比如整数被 0 除：在 X86 平台上，该行为会触发异常，然而在 PowerPC 平台上，该行为不会报错，而是安静地产生未定义的结果。一个完整的 C 语言未定义行为列表可以在 ISO/IEC9899[1](即 C 的标准) 中找到。编译器在对待未定义行为时也有两种做法，一种是**平台相关**，即虽然不同平台会有不同实现，但同一平台上的实现是一致的，另一种则是**未指定行为**，即同一平台上，同一未定义行为的出现场景不一样，结果就会不一致。

编程语言标准之所以申明未定义行为，主要是从该语言的定位出发，不同语言的领域不一样，那么他们的未定义行为也不一样。比如在 JAVA 里缓冲区溢出并不是未定义行为，那么解释器生成出来的目标代码就需要在内存访问指令前面插入边界检查指令。但是这么做会带来性能的损耗，而根据莱斯定理[2]，静态检查出所有缓冲区溢出是不可判定的，只能在运行时做额外检查。C 语言的最大特性就是高效，标准并不想因边界检查引入额外性能损耗，于是直接声明访问越界为未定义行为，所以编译器完全不需要考虑会不会发生访问越界，需要考虑的是使用编译器的人。C 语言的另一大特性的可移植性，在 ARM 上，把一个 32 位整数左移 32 位会产生 0，但是在 x86 上，会产生 1，为了同时支持多种平台，C 语言标准申明这么做是未定义的。另外，Chandler 在 CppCon2016 中指出[3]，狭隘的标准会让语言的定义更加明确，申明未定义行为同时也是在提倡不应该出现这种行为，程序员应该视其为错误，从而帮助程序员写出逻辑更加合理的程序。

但是一方面大部分程序员对不同平台间的微妙差异没有充分的认知，写出来的代码在开发平台上没有问题，但在别的平台上就可能会有问题，甚至是严重的安全漏洞。比如同样的整除在不同平台下行为不一致，导致 PostgresSQL 之前出现了一个安全漏洞[4]，考虑以下用户查询:

```sql
SELECT ((-9223372036854775808)::int8) / (-1);
```

在 32 位系统下，数据库服务器并不会有任何问题，但如果是在 64 位系统下，系统会崩溃，因为 $-2^{63}/-1$ 即带符号整数除法溢出是未定义行为，且在 X86-64 下，IDVI 指令在溢出时会触发异常。

另一方面, 大部分程序员对编译器的优化规则没有充分的认识，写出来的代码在优化前没有问题，但在优化后就可能会出现非常隐蔽的问题，同样也会引起安全漏洞。比如

buf + len < buf 被广泛用来检查指针是否越界（无符号加法溢出了才会更小），在 Python 解释器，Linux 内核等大型系统中都广泛存在着，考虑以下代码片段[5]：

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
   return; /* len too large */
if (buf + len < buf)
   return; /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

因为指针算术运算溢出是未定义行为，当编译器开启了激进的优化选项，会直接把只有当出现未定义行为时才为 TRUE 的条件语句优化成 FALSE，于是对应的分支语句块(安全检查代码) 就会变成不可达代码，被编译器删掉。

除了简化，编译器还会利用未定义行为进行指令重排序，比如：

```
int f(int x) {
   int r = 0;
   for (int i = 0; i < 5; i++) {
      printf("%d\n", i);
      r += 5 / x;
   }
   return r;
}
```

程序员容易认为即使在 x 为 0 时，也至少会输出一个 0，但是编译器的重排序并不会考虑未定义行为，故而认为 r += 5/x 与循环无关，把其移到循环前面，在 x 为 0 时，进入循环前就会报错，并不会输出 0。

这种涉及未定义行为，且在被编译器优化之后，目标代码跟程序员原意不一致，甚至可能导致**安全漏洞**的代码就称之为**不稳定代码**。

对于编译器的这一激进优化，编译器使用者们和编译器开发者们进行过非常激烈的争论。一方面认为这是没有必要的优化，另一方面则认为利用一些很微小的未定义行为能产生很大的性能提升，比如利用 *memory strict aliasing* 能够判断一些对象在循环中不会被修改，从而避免重复从内存读取，又比如[6]，对于 for (int i = 0; i <= N; ++i)，利用“带符号整数加法溢出是未定义行为”，可以断定该循环只会发生 N+1 次，从而进行下一步优化。结合这一系列优化，某些情况下代码甚至能提升接近 10% 的性能。

虽然最终编译器提供一系列编译选项给开发者控制相关的优化规则，比如 -fno-strict-overflow 和 -fno-delete-null-pointer-checks 就会关闭涉及对应未定义行为的优化。但如果只开启了 -O2（很多程序员的默认选项），编译器的激进优化就会导致很多程序员意想不到的结果，甚至安全漏洞。而且不同编译器在不同优化等级下的行为也不一样，相同编译器在不同版本之间也会有一定差异。

1.2 研究背景

虽然为了规避未定义行为带来的风险，程序员应该主动研究清楚语言标准，编译器优化规则，在编写程序时主动避免未定义行为的发生。对于一部分跟编译器前端相关的未定义行为，比如 `(x=2)+(x=3)+(x=3)`，编译器本身就能对开发者提出警告，这些未定义行为也相对地更广为认知，更容易避免。但是对于上一节提到的那些更为微妙的未定义行为，甚至是在更为错综复杂的条件下才会出现的未定义行为，即便是经验丰富的程序员也很难提前察觉。

那么能不能让工具自动化地检测出未定义行为呢？Clang 提供了相关的动态检测工具，即像 Java 一样，在运行时插入额外指令进行检查，但是会伴有性能损耗。而通用的静态分析又是不可判定问题，不过我们可以针对一些主要的未定义行为进行静态检测，只求近似解。目前有两大做法，一是定义一个包含更少未定义行为的形式语义[7]，二是针对不稳定代码，模拟编译器的优化，在发现不稳定现象时，提出警告[4]。这两种做法都存在着相关的实现，不过后一种做法的现有实现已经多年未维护，无法对那些已经使用新版本 Clang 编译器的系统进行分析，这成为了我开始本文相关研究的契机。

1.3 本文工作与结构安排

本文在上述第二种做法的现有算法基础上，基于最新的 LLVM 编译框架、中间语言扫面管理器、可满足性模理论求解器等进行重新设计与实现，并且优化了缓存机制，大大地提高了检查效率，能够对最新的针对 Clang/GCC 编译的系统进行静态分析，发现特定的几类不稳定代码及相关安全漏洞。本文分为六个章节, 大体内容如下:

- 第一章：给出相关研究背景与本文主要工作
- 第二章：介绍静态分析不稳定代码的算法基础
- 第三章：剖析系统整体架构
- 第四章：探讨关键的实现细节
- 第五章：展示在样例代码与 PostgreSQL 源码上的测试结果
- 第五章：对本文进行总结, 并对未来进行展望

第二章　基础知识

本章将简略介绍不稳定代码静态分析的一些基础知识，详细证明可参见参考文献。

2.1　不稳定代码的定义

广义而言，不稳定代码是由于编译器使用者和编译器开发者对未定义行为的认知不一致，在被编译成目标代码之后为跟编写者意图不一致的代码。虽然不同程序员为未定义行为的认知也不一定一样，不过我们可以抽取其中一部分最广泛的共同认知。

首先令符号 C 表示 C 语言的官方标准，令 C' 表示一部分程序员自己认为的 C 语言标准，其中 C' 对未定义行为赋予了特定的定义，其他方面则跟官方 C 标准一致。比如在 C 之下，带符号整数和指针算术运算溢出后是未定义行为，而在 C' 下，带符号整数和指针溢出后会环绕回最小值继续。

令一个代码片段 e 指一个程序源代码 P 中的一个特定的语句或者表达式，令 $P[e/e']$ 表示把 P 中的一个 e 替换为 e' 之后得到的程序源码，那么怎样的 $P \rightsquigarrow P[e/e']$ 转换是合法的呢？

首先不考虑未定义行为，如果 P 和 $P[e/e']$ 对于任意相同的输入都有相同的输出，那么显然这个转换是合法的。接下来考虑未定义行为，把 P 的输入分为两类，一类不会令 P 触发未定义行为，另一类则会，对于第一类输入，P 和 $P[e/e']$ 需要产生相同的输出，但是对于第二类输入，$P[e/e']$ 输出什么都是合法的。注意，一部分在 C 之下是合法的转换，在 C' 之下可能就不合法了，因为 C 中的一部分未定义行为在 C' 中被定义了下来。

定义 (不稳定代码): 程序 P 中的一个代码片段 e 相对于语言标准 C 和 C' 之下是不稳定的，当且仅当存在另一个代码片段 e' 使得 $P \rightsquigarrow P[e/e']$ 在 C 之下是合法的，但是在 C' 之下是不合法的。 ■

2.2　静态检测不稳定代码的算法

以上定义并没有给出一个实际可行的静态检测算法，因为直接考察整个程序的行为是困难且代价高昂的。作为一种近似，我们可以模拟编译器的优化器，且添加一个额外的参数：**是否利用未定义行为进行优化**。首先令优化器不利用未定义行为进行优化，这时发生的转换都是在 C' 之下是合法的。然后令优化器利用未定义行为再进行一遍优化，这时**才**发生的转换，就是在 C' 之下不合法，但在 C 之下合法的，而这些被转换的代码，就是不稳定代码。如果第一次优化有遗漏，且这些遗漏在第二次优化中被转换了，那么就会发生*误报 (False Positives)*。如果第二次优化有遗漏，则会发生*漏报*。接下来将逐步给出“是否利用未定义行为进行优化”这一参数的定义。

2.2.1 良定义程序假设

对于给定的一个代码片段 e，令 $R_e(x)$ 表示它的可达性，该函数返回 `TRUE` 当且仅当 e 在输入为 x 时会被执行。令 $U_e(x)$ 表示 e 的未定义性，该函数返回 `TRUE` 当且仅当 e 在输入为 x 时，会触发未定义行为。比如，对于一个指针解引用语句 `*p`，$U_e(x)$ 为 `p==NULL`，因为空指针解引用在 C 下是未定义行为。

定义 (良定义程序假设): 一个代码片段 e 在输入为 x 时是良定义的，当且仅当 e 不会触发未定义行为，即

$$R_e(x) \rightarrow \neg U_e(x) \tag{2.1}$$

一个程序 P 在输入为 x 时是良定义的，当且仅当 P 中的任意一个 e 都是良定义的，即

$$\Delta(x) = \bigwedge_{e \in P} R_e(x) \rightarrow \neg U_e(x) \tag{2.2}$$

■

2.2.2 消除不可达代码

利用未定义行为，优化器可以消除更多不可达代码。

如果代码片段 e 能被执行当且仅当发生未定义行为，那么它就会被优化器消除掉。因为能令 e 被执行的输入都会令 P 发生未定义行为，对于这些输入，e 能被转换成任意 e'，包括 $\emptyset$，而对于其他输入，$\emptyset$ 跟 e 都不会对 P 的输出产生影响，所以 $P \rightsquigarrow P[e/\emptyset]$ 是合法的。

定理 2.1: 利用未定义行为，优化器可以消除满足以下条件的代码片段 e:

$$\nexists x : R_e(x) \wedge \Delta(x) \tag{2.3}$$

2.2.3 简化不必要运算

利用未定义行为，优化器可以简化更多运算。

如果 e 与 e' 不一致当且仅当发生未定义行为，那么 e 会被优化成 e'。因为能令 e 与 e' 结果不一致的输入都会令 P 发生未定义行为，而对于这些输入，e 能被转换成任意的 e'，所以 $P \rightsquigarrow P[e/e']$ 是合法的。

定理 2.2: 利用未定义行为，优化器可以简化满足以下条件的代码片段 e:

$$\exists e' \nexists x : e(x) \neq e'(x) \wedge R_e(x) \wedge \Delta(x) \tag{2.4}$$

其中 $e(x)$ 为 e 在输入为 x 时的值。

2.3 LLVM 框架

LLVM 是一个自由软件项目、一种编译器基础设施，以 C++ 写成，包含一系列模块化的编译器组件和工具链，可用来开发编译器前端、后端及静态分析工具。

2.3.1 LLVM IR

源代码是结构化的（比如结构化的控制流语句、带作用域的变量），适于程序员编写程序。目标代码又过于扁平（所有控制流都由 jump 控制），适于机器执行程序。而中间语言 *(Intermediate Representation, IR)*[8] 则介于其中，适于静态分析。

LLVM IR 是一种可读的、强类型的、由不受限单赋值寄存器机器指令集组成的 IR。

比如对于以下 C 源码：

```c
int foo(int a)
{
	if (a + 100 < a)
		bar();
	return a;
}
```

编译器前端生成的 LLVM IR 是：

```llvm
define i32 @foo(i32) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = add nsw i32 %3, 100
  %5 = load i32, i32* %2, align 4
  %6 = icmp sgt i32 %4, %5
  br i1 %6, label %8, label %7

; <label>:7:                                      ; preds = %1
  call void @bar()
  br label %8

; <label>:8:                                      ; preds = %7, %1
  %9 = load i32, i32* %2, align 4
  ret i32 %9
}
```

2.3.2 LLVM Pass

在生成 IR 之后，LLVM 的一系列操作都表现为一轮又一轮流水线似的在 IR 上运行的 *Pass*。

并不修改 IR，只能得到相关分析结果的 Pass 叫做 *Analysis*（比如 *Alias Analysis Pass* 能得到变量之间的引用关系），而根据分析结果对 IR 进行原地修改的 Pass 叫做 *Transformation*（比如 *Instruction Conbining Pass* 能合并相邻指令，且后续 Pass 会在简化后的 IR 上进行）。

第三章　系统设计

本章将简略剖析项目的整体架构：在 LLVM 的框架里，对中间语言各级单元（函数、基本语句块、循环、指令）进行一轮又一轮的扫描、转换与分析，参加图3.1

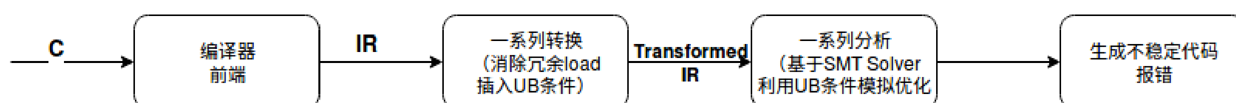

图 3.1　工作流

3.1　生成中间语言

项目的入口是一个脚本程序 `stack-build`，用户只需要把待分析系统的编译命令作为参数传递给该程序，即可自动生成系统所有源程序的中间语言。比如，对于用 `make` 编译的系统，用以下命令生成中间语言：

```
$ stack-build make
```

该程序会介入 `clang` 或者 `gcc` 的前端，添加和删除一些编译参数，并插入编译器前端插件，以控制某些中间语言的生成，减少*误报 (False Positives)*，并且截断目标代码的生成，导出中间语言到指定文件。

3.2　转换中间语言

仅仅介入编译器前端并不能完全生成适于我们后续分析的中间语言，比如，后续分析需要知道*良定义程序假设*会给各个语句带来什么额外的限制条件（即未定义行为条件）。所以要先对中间语言进行多轮的转换。

3.2.1　简化中间语言

在插入未定义行为条件之前，未来提高效率和减少误报漏报，首先要对中间语言进行多方面的简化。

3.2.1.1　简化控制流

LLVM 内置的控制流简化会利用未定义行为，所以我们不能直接调用，而要自己实现一个相似的，但是不利用未定义行为的控制流简化。该轮对每个基本语句块进行以下操作：

- ConstantFoldTerminator
- EliminateDulplicatePHINodes
- MergeBlockIntoPredecessor

3.2.1.2 内联函数

LLVM 内置的函数内联会把内联之后没有调用者的函数消除掉，但是我们的后续分析都是基于函数进行的，所以直接调用 LLVM 内置的函数内联会带来漏报。于是我们自己实现了一个相似的，但是不会消除原函数的内联。

3.2.1.3 消除多余 load 指令

在 LLVM IR 里，`load` 指令用于从内存特定地址中读取值赋给某一变量。而在优化之前，每个变量在每一次使用之前都要通过 `load` 指令从内存中读取值，而不会重用上一次读取的值（即把该值临时保存在寄存器里，下次使用时直接从寄存器快速读取），因为严格来说，同一个变量的两次使用之间，可能会有别的操作把该变量对应的内存的值修改掉。所以在优化之前，LLVM IR 中会有很多冗余的 `load` 指令。

而且 LLVM IR 采用的又是*静态单赋值 (SSA)*[9] 的形式（便于作数据流分析），所以每一个 `load` 指令都会产生一个新的临时变量，这将为后续的转换和分析带来很大的不便，因为我们不能直接通过变量名判断两个变量是否其实是同一个变量。

比如在2.3.1的 IR 中，第 3、5 行的 `load` 语句就是冗余的。

为此，借助 *Memory Dependency Analysis* 和 *Alias Analysis*[10] 的分析结果把每一个基本语句块内冗余的 `load` 指令消除掉，并把这些指令对应的临时变量合并成一个变量，仅保留第一个 `load` 指令。

比如在2.3.1的 IR 中，冗余 `load` 指令会被优化成:

```
%add = add nsw i32 %a, 100, !dbg !17
%cmp = icmp sgt i32 %add, %a, !dbg !17
```

3.2.2 插入未定义行为条件

本阶段会在所有可能发生未定义行为的语句前面插入未定义行为条件，即该语句在什么条件下会发生未定义行为。比如对于 `*p` 语句，就会在它前面插入一个未定义行为条件 `bugon(p==nullptr)`。在实际实现时，`bugon()` 函数的函数体为空即可，该函数调用语句仅仅用于存放未定义行为条件，供后续分析使用。针对不同类别的未定义行为，会有专门的扫描函数添加对应的未定义行为条件，在添加完全部未定义行为条件之后，才会开始进行分析。

比如在2.3.1的 IR 中，对于 `ICMP` 指令（对应着 `if (a + 100 < a)` 语句），这一轮转换会在它前面插入 `bugon(sadd.overflow(a, 100))`:

```
%0 = call { i32, i1 } @llvm.sadd.with.overflow.i32(i32 %a, i32 100)
%1 = extractvalue { i32, i1 } %0, 1
call void @opt.bugon(i1 %1), !dbg !13, !bug !15
%cmp = icmp sgt i32 %add, %a, !dbg !17
```

3.3 分析中间语言

这一步将分别不利用和利用未定义行为条件分析目标语句是否可以被优化，并认为利用了未定义行为条件（即在*良定义程序假设*之下）才能被优化的代码为不稳定代码，向用户提出警告。

3.3.1 检测会被消除的不稳定代码

判断语句相对于程序起点是否可达的代价过于高昂，所以实现上我们近似成判断相对于所在函数起点是否可达。对于每一个基本语句块进行扫描，如果不考虑未定义行为条件，该语句块就已经不可达了，直接跳过。否则，再把未定义行为条件考虑进去，如果此时该语句块才变为不可达，就认为这个基本语句块包括不稳定代码。

3.3.2 检测会被简化的不稳定代码

目前支持两种类别的检测：

- 布尔简化：在*良定义程序假设*之下才为 `TRUE` 或者 `FALSE` 的布尔表达式
- 代数简化：在*良定义程序假设*之下才能进行不等式简化的整数比较表达式

比如在2.3.1的 IR 中，对于 `ICMP` 指令（对应着 `if (a + 100 < a)` 语句）：

`a + 100` 溢出之后会小于 `a`，所以一开始并不能进行数学上的不等式简化，即在两边同时删除 `a` 使其变成 `100 < 0`。

可是在*良定义程序假设*之下，考虑上 `bugon(sadd.overflow(a, 100))` 这个未定义行为条件，就能把它简化成 `100 < 0`，进而简化为 `FALSE`，最终把整个语句块都消除掉。

第四章　系统实现

本章将探讨一部分较为关键的具体实现细节。

4.1　介入编译过程

待分析的系统往往都有着很复杂的编译配置、编译命令，而且各不相同，不能为其一一定制。所以采用了一个更具通用性的做法：首先替换环境变量，使得系统编译时调用的 clang, cc1 等程序变成我们的脚本，然后脚本根据接收到的编译参数，生成对应的中间语言，并且可以根据需要插入编译器前端插件。

4.2　中间语言扫描

项目基于 LLVM 的 *New Pass Manager*[11] 对 IR 进行扫描，*New Pass Manager* 是一个基于 *concept-based polymorphism*[12] 的管理器，被管理的 *Pass* 并不需要继承某个特定的基类，只需要实现 name 方法与 *run* 方法即可。然后再结合 *Pass Builder* 即可构建一条 *Pass Pipeline*，像流水线一样对 IR 的各级单元进行扫描、转换、分析。

4.2.1　BugOnPass

添加未定义行为条件的一系列 *Pass* 的基本架构是一样的，只是因针对不同的未定义行为而在实现上有部分区别。所以把他们的共同特征抽象成一个 BugOnPass 基类，该类封装了 void setInsertPointAfter(llvm::Instruction *) 和 llvm::Value *createIsSAddWrap(llvm::Value *, llvm::Value *) 等与插入 bugon() 未定义行为条件相关的底层函数。在继承该基类的基础之上，只需要添加少量代码即可完成对一类未定义行为的支持。

4.2.2　BugFreePass

同理，分别不利用和利用未定义行为条件模拟优化器的那一系列 *Pass* 也基于一个基类 BugFreePass(*BugFree* 是良定义程序假设的缩写)。该类封装了 llvm::SMTExprRef getBugFreeDelta(llvm::BasicBlock *) 和 llvm::Optional<bool> query-WithBugFreeDelta(llvm::SMTExprRef E, llvm::SMTExprRef Delta) 等与搜集利用未定义行为条件相关的底层函数。

4.2.3 Diagnostic

该类用于输出不稳定代码位置和相关的未定义行为。单纯把不稳定代码附近的未定义行为都输出的话，会给用户增加很多无关干扰。为了得到导致不稳定的核心因素，我们采取贪心的做法，对所有未定义行为逐个遮盖，判断它的缺失是否会导致不稳定代码变稳定，即可得到最小化的不可或缺的未定义行为集合。

4.3 可满足性模理论求解

项目基于 *SMT Solver*[13] 对 $\Delta(x)$、$R_e(x)$ 和 $U_e(x)$ 等条件及相关公式进行求解。比如对于整数相关的指令，可以用 *Bitvector* 进行建模，一个 IR AST 可以生成一个对应的 *SMT Expr AST*。需要注意的是，在 C 里，布尔型是可以和整数一起参与整数运算的，而在 SMT 里，需要先手动把布尔型转成 0/1 *Bitvector*。

4.3.1 统一接口

目前业界存在着很多优秀的 *SMT Solver* 可供调用，他们在不同的方面各有优劣，可是在 API 方面也有着一定区别。为了方便替换及比较，我们并不直接使用它们提供的 API，而是定义一个统一的接口文件 `SMTLIB.h`，不同的 *SMT Solver* 有不同的接口实现。在进行了这样的一层封装之后，项目其他地方只能调用 `SMTLIB.h` 中的接口，在需要替换求解器后端的时候只需要替换接口实现即可。

4.3.2 简化求解

虽然我们可以迅速地搜集任意范围代码包含的未定义行为条件，不过过多的限制条件会令 *SMT Solver* 求解过慢，所以对于每一个待分析的目标语句 e，把 $\Delta(x)$ 的范围从整个程序缩小成 e 的控制流支配者到 e 之间，可以证明，简化公式跟原公式是等价的。

4.3.3 缓存结果

我们需要经常性地获取某个语句或者是某个基本语句块的可达性条件，这其中存在着大量的重复，所以缓存是非常关键且必要的。`ValueGen` 类用于获取指定语句的可达性条件（结果为 *SMT Expr AST*），在此基础之上，`PathGen` 类用于获取指定基本语句块的可达性条件。这两个类都有一个基于二次探查散列表 (*Dense Map*) 实现的 `Cache` 成员变量，每当一个 `Value` 或者一个 `Path` 的 *SMT Expr AST* 构造出来，都会把结果存放到这个 `Cache` 中，在下次查询时，如果 `Cache` 里面已经有结果了，会直接返回缓存结果，否则才会进行构造。

第五章　系统测试

本章将展示该静态分析系统的测试结果，主要从三个方面进行考量：效率、误报率、报告结果是否准确可读。

5.1　样例测试结果

项目本身内置了一系列不稳定代码用作测试用例。这些测试用例并非凭空虚造，均是在网上从别的项目里搜集出来，再进行简化的不稳定代码。比如以下测试用例：

```
int foo(unsigned char log_groups_per_flex) {
	unsigned int groups_per_flex;
	groups_per_flex = 1 << log_groups_per_flex;
	if (groups_per_flex == 0) {
		bar();
		return 1;
	}
	return 0;
}
```

便是从 Linux 内核的 BUG 页面`https://bugzilla.kernel.org/show_bug.cgi?id=14287`里摘出来的。

对于该样例，首先用 `stack-build` 把它编译成以下中间语言：

```
define dso_local i32 @foo(i8 zeroext %log_groups_per_flex) #0 !dbg !7
    {
entry:
  %retval = alloca i32, align 4
  %log_groups_per_flex.addr = alloca i8, align 1
  %groups_per_flex = alloca i32, align 4
  store i8 %log_groups_per_flex, i8* %log_groups_per_flex.addr, align
     1
  %0 = load i8, i8* %log_groups_per_flex.addr, align 1, !dbg !13
  %conv = zext i8 %0 to i32, !dbg !13
  %shl = shl i32 1, %conv, !dbg !14
  store i32 %shl, i32* %groups_per_flex, align 4, !dbg !15
  %1 = load i32, i32* %groups_per_flex, align 4, !dbg !16
  %cmp = icmp eq i32 %1, 0, !dbg !18
  br i1 %cmp, label %if.then, label %if.end, !dbg !19

if.then:                                          ; preds = %entry
  call void @bar(), !dbg !20
  store i32 1, i32* %retval, align 4, !dbg !22
  br label %return, !dbg !22

if.end:                                           ; preds = %entry
  store i32 0, i32* %retval, align 4, !dbg !23
  br label %return, !dbg !23

return:                                           ; preds = %if.end, %if.then
  %2 = load i32, i32* %retval, align 4, !dbg !24
  ret i32 %2, !dbg !24
```

接着用 `stack` 静态分析这段代码，将得到以下 `YAML` 格式的报告：

```
---
bug: bugfree-dce
model: |
  %cmp = icmp eq i32 %shl, 0, !dbg !17
  --> true
  ************************************************************
  if.end:
  store i32 0, i32* %retval, align 4, !dbg !22
  br label %return, !dbg !22
stack:
  - /home/guangqing.chen/stack/build/test/testcases/ext4shl.c:15:2
ncore: 1
core:
  - /home/guangqing.chen/stack/build/test/testcases/ext4shl.c:10:22
    - shift left overflow
```

bug 项 表示在*良定义程序假设*之下，对应代码会被优化器消除 *(Dead Code Elimination, DCE)*

mode 项 不仅会输出对应不稳定代码的中间语言，还会在其基础上补充额外信息，提高可读性，方便用户 Deubg。比如在该例子里，`model` 项形象地指出了在*良定义程序假设*之下，`%cmp` 变量会被优化成 `TRUE`，使得 `if.end` 语句块不可达。

stack 项 输出报错代码所在文件路径，及该文件的调用栈。

core 项 用来报告导致代码不稳定的未定义行为，比如在该例子中，`ext4shl.c` 文件的第 10 行会触发左移溢出这一未定义行为，导致第 15 行被 `DCE`。可见 `YAML` 格式的报告非常简洁可读，未来还可以在此基础上开发工具高效解析该纯文本报告，生成更可读的报告（比如网页）。

5.2 PostgreSQL 测试结果

PostgreSQL 是一款主流的开源数据库，忽略空行和注释，整个 PostgreSQL (V11.2) 源码树包含了 847921 行 C 代码。其中核心的 `backend` 模块包含了 609387 行 C 代码，该模块源码生成出来的中间代码多达 1000 万行，在 `Intel(R) Xeon(R) W-2123 CPU @ 3.60GHz` 环境下，对其进行静态分析耗时 27 分钟 24 秒，发现了 4 个在*良定义程序假设*之下会被代数简化的不稳定代码，分别位于 `utils/adt/varbit.c:1064`，`utils/adt/varlena.c:838`，`utils/adt/varlena.c:902`，`utils/adt/varlena.c:2909`。尽管还未确定这几处漏洞是否可被外部触发，但表明了本系统是高效且可用的。

第六章　总结与期望

6.1　总结

本文系统地阐述了研究背景、理论基础、项目整体架构、部分实现难点与测试结果，若想要更进一步地了解，可参见代码`http://github.com/gou4shi1/stack` (包含了 4000 行左右的 C++ 代码)

原有的实现多年未维护，已经无法分析使用新版本 Clang 编译的系统，该领域也没有出现针对不稳定代码的替代品，因此本实现将会是一个有力的补充，有助于减少相关的安全漏洞的出现。

由于使用了大量还在活跃开发并不稳定且没有文档的 LLVM 新特性，项目开发遭遇了很大的困难，最终通过阅读 LLVM 的源码，甚至是参与 LLVM 的开发才得以解决。

除了使它重焕活力之外，通过 *New Pass Manager* 在每一轮扫描之后精密地控制以往的分析结果是否缓存，针对 *SMT Expr AST* 的构造进行高效的缓存，等等的实现细节也令它的效率得到了很大的提高。

6.2　展望

虽然大致上本项目实现了很高的完成度，但是依然存在着很多改进空间:

- 跟进 LLVM 的开发进展，对本项目进行持续更新，保持活力
- 添加对更多不稳定代码的支持，比如利用未定义行为进行重排序优化
- 改进报错界面，帮助用户更快捷地修复问题
- 结合更精密的形式语义，探索进一步的发展空间

参考文献

[1] Programming languages −C [M]. 2011.

[2] Pinkus A. A simple proof of the Hobby-Rice theorem [J]. Proceedings of the American Mathematical Society, 1976, 60 (1): 82–84.

[3] Carruth C. Garbage In, Garbage Out: Arguing about Undefined Behavior with Nasal Demons [C]. 2016.

[4] Wang X, Zeldovich N, Kaashoek M F, et al. Towards optimization-safe systems: Analyzing the impact of undefined behavior [C]. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013: 260–275.

[5] Wang X, Chen H, Cheung A, et al. Undefined behavior: what happened to my code? [C]. In Proceedings of the Asia-Pacific Workshop on Systems, 2012: 9.

[6] Lattner C. What every C programmer should know about undefined behavior [J], 2011.

[7] Hathhorn C, Ellison C, Roşu G. Defining the undefinedness of C [C]. In ACM SIGPLAN Notices, 2015: 336–345.

[8] Chisnall D. Modern Intermediate Representations (IR) - LLVM [C/OL]. 2017. https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf.

[9] Cytron R, Ferrante J, Rosen B K, et al. Efficiently computing static single assignment form and the control dependence graph [J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1991, 13 (4): 451–490.

[10] Whaley J, Lam M S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams [J]. ACM SIGPLAN Notices, 2004, 39 (6): 131–144.

[11] Carruth C. THE LLVM PASS MANAGER PART 2 [C]. 2014.

[12] Parent S. Inheritance Is The Base Class of Evil [C]. 2013.

[13] De Moura L, Bjørner N. Z3: An efficient SMT solver [C]. In International conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008: 337–340.

致 谢

感谢指导老师王立斌从大一到大四对我的指导，祝老师身体安康。
感谢 Wang Xi 和 Zeldovich Nickolai 等前辈的成果，对本论文有着很大的启迪。
感谢 LLVM/Clang 社区的热心开发与积极回复，使本项目顺利完成开发。
感谢 Github 上诸位师兄相继维护的 SCNUThesis LaTeX 模板，让本论文撰写顺利。
最后要感谢我的家人及许坤对我的大力支持。