目录

1 多语言程序的内存错误与指针分析

1.1 多语言程序的内存错误

High-level 的宿主语言一般采用自动垃圾收集机制管理其堆内存。例如，Python 使用引用计数算法进行自动垃圾收集，Python runtime 在程序执行过程中调整引用计数并维持对象的引用计数（ob_refcnt）等于其引用数的不变式，当对象的引用计数为 0 时，垃圾收集器回收其堆空间。Go 使用优化的标记清除算法进行自动垃圾收集 [Go 内存模型, Go 垃圾收集器，Go 垃圾回收算法]，当堆分配占用量或回收时间间隔达到触发阀值，则会进行标记清除以回收不可达的内存占用。

然而，外部语言一般不在宿主语言的垃圾收集器的管理范围内。在多语言程序中，当底层模块通过外部函数接口（FFI）操作宿主语言对象时，会影响宿主语言对该对象的自动回收。例如，当底层 C 模块通过 Python/C API 操作 Python 对象时，Python runtime 不会自动调整其引用计数。Go 通过 cgo 调用将 Go 对象传入底层 C 模块后，Go runtime 无法知道底层模块对该对象的持有周期以及是否会修改其内存，底层模块也无法知道传入的内存是在堆上还是栈上分配的，是否需要释放 [cgo 内存管理]。

Listing 1: 错误的 Python/C 接口程序

```
1  PyObject *test(PyObject *self, PyObject *args)
2  {
3    PyObject *list;
4    PyObject *item;
5    list = PyList_New(1);
6    if (!list)
7      return NULL;
8    item = PyLong_FromLong(42);
9    if (!item) {
10     /* Memory leak: missing Py_DECREF(list) */
11     return NULL;
12   }
13   /* This steals a reference to item; item is not leaked. */
14   PyList_SetItem(list, 0, item);
15   return list;
16 }
```

由于 Python runtime 不能自动调整底层 C 模块中的 Python 对象的引用计数，底层模块需要通过增减对象引用计数的 Python/C API 来手动维护引用计数，这是易错的。Listing 1 所示为包含内存错误的 Python/C 接口代码，`PyList_New` 执行成功会将 `list` 的引用计数加 1，在函数返回前，需要手动将其引用计数减 1 以避免内存泄漏。“偷引用”等特性 [偷/借引用] 的存在（如 Listing 1 中的 `PyList_SetItem`，对比 `PyList_Append` [StackOverflow 错误例子]）增加了人工管理引用计数的难度。

Listing 2: 错误的 cgo 程序

```
1  package main
2  /*
3  #include <stdio.h>
4  void cFuncChar(char* ptr) {
5    printf("%s\n", ptr);
6  }
7  void cFuncVoid(void* ptr) {
8    printf("%s\n", (char*)ptr);
9  }
10 */
11 import "C"
12 import "unsafe"
13 type MyStruct struct {
14   Distraction [2]byte
15   Dangerous *int
16 }
17 func main() {
18   cstr := C.CString("hello")
19   /* Memory leak: missing C.free(cstr) */
20   bypassPanic(1)
21   triggerPanic(1)
22 }
23 func bypassPanic(i int) {
24   ms := &MyStruct{[2]byte{'A', 0}, &i}
25   C.cFuncChar((*C.char)(unsafe.Pointer(ms)))
26 }
27 func triggerPanic(i int) {
28   ms := &MyStruct{[2]byte{'B', 0}, &i}
29   C.cFuncVoid(unsafe.Pointer(ms))
30 }
```

Go 的垃圾收集器不能自动回收底层 C 模块中的 Go 对象，需要底层模块通过 cgo 调用手动回收，同时还面临其他问题。

1. 标记清除算法（可能存在内存搬运，栈扩容也可能）相较引用计数算法的复杂性引入了额外约束 [cgo 传指针的约束]，cgo 不应向外部函数传递一个指向包含 Go 指针的内存的 Go 指针 [cgoCheckPointer 实现]，Listing 2 第 29 行在运行时会 panic。

2. 聚合类型与矢量类型，聚合类型如 struct 的域、矢量类型如 array 和 slice 的整体应满足 1 中的约束。

3. cgo 程序常常用到 unsafe 特性，比如为了避免语言间的内存拷贝，但是 C 代码直接操作 Go 内存的安全性需要程序员自己保证。

4. 底层模块应维持 1 中的约束，C 代码不应向 Go 内存中放入 Go 指针，哪怕是临时地。Listing 2 第 25 行的代码通过 Go 指针向 C 指针的类型转换绕过了运行时检查 [cgo 绕过指针检查]，但是此时 C 代码对该内存的修改或回收都可能导致安全问题。

5. 多层指针与地址操作符。

6. 已知的实现上的漏洞 [Go issue 28606]，向 C 内存中写 `nil` 或 C 指针是允许的，但是存在把 C 内存误认为 Go 指针，进而误报 1 中约束违反的问题；同时全局变量的使用也可能导致漏报 1 中约束违反的问题，Listing 2 中把变量 `i` 改成全局变量则第 29 行的代码也无法得到运行时报错（更新：应为"一个指向包含 Go 指针的堆内存的 Go 指针"，全局变量在数据区）。

1.2 已有的解决方案

1.2.1 gcc-python-plugin/gcc-with-cpychecker

gcc-python-plugin 是一个链接了 libpython 的 GCC 插件 [GCC plugin]，并在此基础上实现了一个静态检查工具 CPyChecker [code, doc]，检查 CPython 的 C 扩展模块中引用计数和异常状态相关的漏洞。

CPyChecker 对引用计数的跟踪包含两个部分，函数的上下文内可知的引用计数和程序其他部分持有的引用。例如在 Listing 1 中，`PyList_New(1)` 调用成功后，`list` 对象的引用计数记为 $1+N$，其中 N 是程序其他部分持有的引用的未知值，$N \geq 0$。CPyChecker 通过函数的上下文结束后对象的引用计数是否为 $0 + N'$ 来检查内存错误，对象被存储到其他未知位置可能导致 $N' = N + 1$。

CPyChecker 存在如下不足。

1. 对于循环，只跟踪其一次迭代。

2. 跟踪路径数有上限限制，不能分析复杂的程序。

3. 缺乏函数间的分析，它假设只要函数返回 Python 对象（`PyObject*`），必然是一个新的引用或 `NULL`（异常情形），函数间的"偷/借"引用行为需要人为指定。

4. 基于 GCC 的别名分析 [alias analysis, tree SSA passes]，能够检查一些简单的空指针解引用的问题，但是不支持域敏感的分析 [release note 0.7] 和多层指针 [issue 167, `PyObject *PyLong_FromString(const char *str, char **pend, int base)`]。

5. 不支持弱引用 [issue 58, 弱引用 Python/C API]。

1.2.2 Li & Tan 2014 Pungi

Pungi 是 Li 和 Tan 在 ECOOP 2014 提出的一种检查 Python/C 接口代码中的引用计数内存错误的方法。其核心思路是：在变量的引用不逃逸出其作用域时，一个对象引用计数的变化量在其作用域结束时应该为零；在存在逃逸时（通过返回值或写堆内存），则应该等于其引用的逃逸数。Pungi 利用仿射抽象（affine abstraction）来跟踪这一变化量，通过程序变换把赋值语句的 right-hand side 变成仿射形式 $a_0 + \Sigma_{i=1}^{n} a_i x_i$，其中 a_i 是常数，x_i 是变量。Figure 1 是一个带有逃逸引用的仿射变换例子，仿射程序变量 `rc` 记录引用计数变化量，`on` 是对象非空的标记变量，`ev` 记录变量的引用逃逸数。

```
PyObject *foo() {
  PyObject *pyo = PyInt_FromLong(10);
  if (pyo == NULL) {
    return NULL;
  }
  return pyo;
}
```

```
foo() {
  locals rc1, ev1, on1;
  rc1 = 0; ev1 = 0;
  if (?) { rc1 = 1; on1 = 1; }
  else { rc1 = 0; on1 = 0; }
  if (on1 == 0) {
    assert (rc1 == ev1);
    return;
  }
  if (on1 == 1) ev1++;
  assert (rc1 == ev1);
  return (rc1, ev1);
}
```

图 1: 一个带有逃逸引用的仿射变换例子

Pungi 通过基于 SSA 形式的变换、对引用逃逸的补充和过程间分析解决了 CPyChecker 的不足 1、2、3，但它也存在以下不足。

1. 利用仿射程序分析引用计数基于 shallow alias 的假设，即多层引用一定不是别名。
2. 由于缺少宿主语言的指针信息，只能假设底层入口函数的参数对象不会引用同一对象。

1.2.3 Mao et al. 2016 RID

RID 是 Mao 等人在 ASPLOS 2016 提出的检查引用计数错误的方法，它把引用计数的定义做了一个扩展。

1. 一个动态分配对象的引用计数，是对指向该对象的指针数量的跟踪。
2. 一个设备结构的引用计数，是对使用该设备的线程数的追踪。

从而把引用计数错误检查扩展到了使用定义 2 的 Linux 内核。

Listing 3: 错误的 Linux 电源管理子系统代码

```
int reg_read(device *d, int reg);
void inc_pmcount(device *d)
int foo(device *dev) {
  assert(dev != NULL);
  int v = reg_read(dev, 0x54);
  if (v <= 0)
    goto exit;
  inc_pmcount(dev);
  // more register reads/writes
exit:
  return 0;
}
```

Listing 3 是 Linux 电源管理子系统的错误例子，它也显示出 CPyChecker 和 Pungi 这种跟踪作用域内引用计数变化量的方法的一个缺点，即需要知道引用计数 API 的语义，这对于基本固定且大小适中的 API 集合而言是可行，例如 Python/C API 的引用计数子集；但对于 Linux 内核中众多且变化的子系统而言，引用计数 API 可能很多且处于迭代变化中。为此，RID 采用了一种不同的引用计数错误检查方法，称为不一致路径对（inconsistent path pairs, IPP）。Listing 3 对应的不一致路径对如 Figure 2 所示。假设，在运行时，程序到达 `foo` 函数的入口时，`dev` 的 PM 计数为 n，如果执行 p_1 路径，则 PM 计数变为 $n+1$，而执行 p_2 路径则是 n，(p_1, p_2) 为不一致路径对。

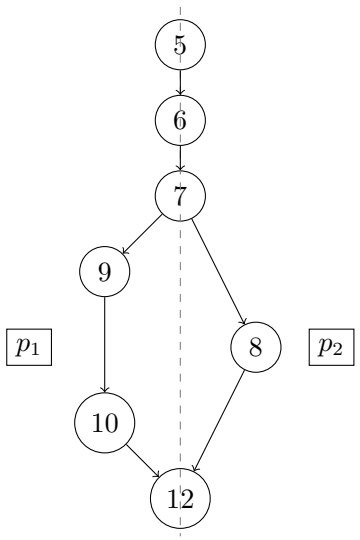

图 2: Listing 3 例子对应的不一致路径对

RID 存在如下不足。

1. 扩展定义部分，Linux 内核子系统的引用计数 API 的语义基于 'inc'-'dec'、'get'-'put' 这样的关键字匹配，评估部分没有明确说误报率，根据文中数据人工算了一下高达 77%。
2. 非扩展定义部分，Python/C API 的语义仍需人工维护，但是只关注增减，而不关注是对哪个参数对象的增减，只提供了少量评估数据，并且是和 CPyChecker 对比，而不是和 Pungi 对比，提升与 Pungi 类似地来源于 CPyChecker 不是基于 SSA 形式的。交叉对比两篇文章的数据，RID 的精度应该不如 Pungi。
3. 循环只展开一次。
4. 分析路径数有上限。
5. 不依赖指向分析且只支持线性算术，无法处理函数指针、向数据结构的域中存储和位操作。

1.2.4 Kondoh & Onodera 2008 BEAM-JNI

BEAM(Brand 2000)是由 IBM 设计的一个多种 Java 程序漏洞的检查工具,Kondoh 和 Onodera 在 ISSTA 2008 为 BEAM 增加了对 4 类 JNI 漏洞的检查 BEAM-JNI，其中两类与内存相关，一个是与 Listing 2 第 19 行类似的内存泄漏（及多次回收和回收后使用），一个是使用无效的局部引用（Listing 4）。

Listing 4: 错误的 JNI 程序

```
static jclass fooCls;
JNIEXPORT void JNICALL Java_Foo_initialize(JNIEnv *env, jclass cls) {
  fooCls = (*env)->FindClass(env, "Foo");
  /* fooCls should be converted into global: missing
     fooCls = (*env)->NewGlobalRef(env, fooCls); */
}
JNIEXPORT void JNICALL Java_Foo_bar(JNIEnv *env, jobject object) {
  jmethodID mid = (*env)->GetMethodID(env, fooCls, "foo", "()V");
  /* Memory leak: missing env->DeleteGlobalRef(fooCls); */
}
```

JNI 对象 [JNI 与 GC]，即底层 C 代码中的 Java 对象，以引用形式存在，Table 1 是其三种不同的引用方式，本地引用的作用域仅限于其创建栈帧和线程，创建栈帧返回时则自动删除；把本地引用提升为全局引用后，则可以在附属于 JVM 的任何线程中使用，但也需要被手动回收；弱引用则是一种由 GC [JVM 复制垃圾收集器] 负责回收的全局引用，只要一个弱引用对象没有任一强引用，GC 就可能在任意时候回收它，并使得引用变成空值，该引用只能被回收，所有后续使用都需要先提升为强引用。

表 1: JNI 对象引用

JNI 对象引用	GC 自动回收	跨函数/跨线程	作用域自动释放
本地引用（local reference）	N	N	Y（也可以手动释放）
全局引用（global reference）	N	Y	N
弱引用（weak reference）	Y	Y	N

BEAM-JNI 存在以下不足。

1. 不支持检查全局引用的内存泄漏。
2. 对于无效的局部引用错误，当一个全局变量被多次赋值且是被全局引用重写时，BEAM-JNI 会产生误报。
3. JNI 函数调用是间接的通过指针进行的（与反射机制类似），BEAM-JNI 不知道底层调用点被实际调用的 Java 函数。比如对于异常处理错误，就只能假设所有函数都可能抛出异常，从而产生误报。

1.2.5 与指针分析的关系

以上多语言内存错误检查工具在宿主语言和外部语言侧是否使用了指针分析？如果使用了，那么指针分析的使用和工具的内存错误的分析是否是解耦的？增强使用的指针分析或接入指针分析（$\bowtie$ 表示需要跨语言的指针分析）是否可以提升内存错误检查的精度？

表 2: 已有的解决方案和指针分析的关系

内存错误检查	指针分析			
	宿主语言	外部语言	解耦	提升
CPyChecker	✗	✓	✓	✓（不足 3、4）
Pungi	✗	✗	-	✓（不足 1、$2^{\bowtie}$）
RID	✗	✗	-	✓（不足 5）
BEAM-JNI	✗	✗	-	✓（不足 $3^{\bowtie}$）

1.3 案例分析

考虑宿主语言采用某种垃圾收集机制，外部语言依赖显式手动回收的情形。

1. **宿主侧释放宿主语言对象**通过 GC 自动进行。
2. **宿主侧释放外部语言对象**通过外部函数调用进行，例如 `C.free()`。
3. **外部侧释放外部语言对象**通过手动内存管理进行，例如 `free()`。
4. **外部侧释放宿主语言对象**通过外部函数接口 FFI 手动管理，例如 `Py_DECREF()`、`DeleteGlobalRef()`。

前述的问题和解决方案主要关注第 4 类可能导致的内存泄漏。更一般地，我们考虑以上各类在多语言程序中可能存在的多种内存错误，以及需要的指向信息。

1.3.1 Use-after-free

根据语言侧的不同，分为：

- 外部侧分配的内存（外部语言对象），在外部侧被 free 后（以上第 3 类），在宿主侧 use。
- 宿主侧分配的内存（宿主语言对象），在外部侧被 free 后（以上第 4 类），在外部侧 use。
- 宿主侧分配的内存（宿主语言对象），在外部侧被 free 后（以上第 4 类），在宿主侧 use。

Use 包括：(i) 解引用（悬空指针解引用），(ii) 再次 free（多次回收）。

1.3.2 空指针解引用

Free 之后置空可以避免再次 free 触发错误，但是对空指针解引用会发生段错误。

- 外部侧分配的内存（外部语言对象），在外部侧被 free 并置空后，在宿主侧解引用。
- 宿主侧分配的内存（宿主语言对象），在外部侧被 free 并置空后，在外部侧解引用。
- 宿主侧分配的内存（宿主语言对象），在外部侧被 free 并置空后，在宿主侧 use。

1.3.3 指针未初始化

对仅声明未初始化的指针解引用是悬空指针解引用的特殊情况。

- 在宿主侧声明的指针，在外部侧解引用。
- 在外部侧声明的指针，在宿主侧解引用。

1.3.4 其他约束

如 cgo 要求 C 侧不应持有一个指向包含 Go 指针的堆内存的 Go 指针（见 Listing 2 及相关讨论）。

1.4 相关的单语言工作

指针分析是一个高复杂度的基础分析，虽然可以优化多种上层分析，但在已有的分析中集成指针分析往往存在开销过大的问题。然而，指针分析作为一种描述内存关系的分析框架，在需求驱动的指针分析中集成其他分析是一种可能的解决方案（异常处理 [1, 2]，内存错误检查 [5, 4]，调用图构建 [3]）。

1.4.1 Wang et al. 2019 TsmartGP

TsmartGP [4] 是 Wang 等人在 ASE 2019 提出的一种利用指针分析检查指针未初始化的内存错误的方法，通过流敏感、上下文敏感、准路径敏感提升指针分析精度，并通过多入口机制优化效率。如 Listing 5 所示，考虑 else 分支，`p` 的指向集合可能未空，对其解引用会导致悬空指针解引用。

Listing 5: 指针未初始化错误例子

```
1 int main() {
2   int *p;
3   int a;
4   scanf("%d", &a);
5   if (a > 0)
6     p = &a;
7   *p = 0;
8   return *p;
9 }
```

Cppcheck 和 Clang Static Analyzers 中也集成了指针相关的分析，但是对未初始化的错误误报率分别高达 96.9% 和 83.3%，TsmartGP 有效通过更精确的指针分析优化了内存错误检查的精度。

1.4.2 Yan et al. 2018 CRed

CRed [5] 是 Yan 等人在 ICSE 2018 提出的一种利用指针分析检查 use-after-free（UAF）错误的方法。假设有一对表达式 $(free(p@l_f), use(q@l_u))$，其中 p 和 q 为指针，l_f 和 l_u 为行号。记 $\mathcal{P}(l)$ 为从 main 到 l 的所有可能路径组成的集合。那么该表达式对是 UAF 漏洞当且仅当时空相关性 $\mathbb{ST}(free(p@l_f), use(q@l_u))$ 成立：

$$\mathbb{ST}(free(p@l_f), use(q@l_u)) := \exists(\rho_f, \rho_u) \in \mathcal{P}(l_f) \times \mathcal{P}(l_u) : (\rho_f, l_f) \rightsquigarrow (\rho_u, l_u) \wedge (\rho_f, p) \cong (\rho_u, q)$$

$\rightsquigarrow$ 表示**时间可达**（在程序的过程间控制流图上），$\cong$ 表示**空间别名**（p 和 q 指向同一对象），(ρ, l) 表示路径 ρ 下的程序点 l。

CRed 首先使用一个快速但不精确的指针分析得到备选的 UAF 对，然后做两次基于需求驱动的指针分析的规约来精化结果，一次是上下文规约，一次是路径规约。

[1] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–12, 2009.

[2] George Kastrinis and Yannis Smaragdakis. Efficient and effective handling of exceptions in java points-to analysis. In *International Conference on Compiler Construction (CC)*, pages 41–60, 2013.

[3] Yannis Smaragdakis, George Balatsouras, et al. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

[4] Yuexing Wang, Guang Chen, Min Zhou, Ming Gu, and Jiaguang Sun. TsmartGP: A tool for finding memory defects with pointer analysis. In *International Conference on Automated Software Engineering (ASE)*, pages 1170–1173, 2019.

[5] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *International Conference on Software Engineering (ICSE)*, pages 327–337, 2018.

1.4.3 距解决多语言问题的差距

仅考虑两个语言共享地址空间的互操作机制（通过网络通信的方式除外），假设我们已经分别有宿主语言和外部语言的单语言的指针分析方法，我们距离得到多语言程序的指向关系还有哪些方面的问题需要解决?

数据类型 指针分析中说的指针，并不局限于 C、C++、Go 等语言的指针数据类型，而是指所有引用类型。区别于原子类型，引用类型的值之间可以共享状态。例如，可以把 Java 指针分为局部变量、静态字段、实例字段、数组元素 4 类。而带有指针数据类型的语言，往往需要考虑多层指针的问题。多语言环境下需要解决如何兼容不同语言的指针定义，以及如何表示它们之间的引用关系。

求值策略 在 C 语言中，参数是按值调用的，而高层语言可能采用不同的求值策略，如按引用调用、按名调用等。当参数在两个语言间传递时，一个常见的策略是在接口代码中利用指针模拟按引用传递，而接口代码可能是使用 FI 手动编写或（部分）自动生成或编译期动态构建的。多语言环境下需要解决如何兼容不同的求值策略。同时需要注意，单层指针和按引用调用一起可以模拟多层指针，这样的组合行为可能使得问题更加复杂。

内存抽象 对于 C/C++ 等语言，指针分析回答指针指向的内存位置；对于 Java 等面向对象语言，指针分析回答指针指向的对象。而内存位置或对象往往基于某种堆抽象，如常见的分配点抽象。多语言环境下需要解决如何兼容不同的内存抽象，同时考虑多语言程序的内存布局，尤其对于内存错误检查驱动的指针分析。同时，原子类型例如字符串由于数据布局的不同，在接口层可能发生内存复制。

类型转换 值在跨语言传递时，除了需要兼容求值策略，还需要考虑类型转换。由于数据布局的异同，当存在兼容数据类型时，语言间往往存在一个非一对一的类型映射关系。例如，Python 的长整型通过 PyLongObject 转换到不同长度的 C 整型。考虑数据类型，包括推导类型如指针和函数，组合类型如结构体和数组，自定义类型等，类型转换可能带来指向关系的变化。多语言环境下需要考虑类型转换带来的影响。

敏感性选择 敏感性选择关乎指针分析的精度和效率，但是不同特性的语言往往需要不同敏感性的指针分析。例如，对于命令式语言，流敏感带来的精度提升大于上下敏感，这在面向对象语言上则相反。同时，不同的语言侧可能采用不同的上下文敏感方法和深度，已有的可选上下文方法是基于单语言的。多语言环境下需要解决语言特性对敏感性效果的影响，解决如何兼容两种不同语言上的不同敏感性，以及对混合精度下限的评估。

1.5 总结

- 已有的多语言内存错误检查工作大多不依赖指针分析，但是可以从指针分析得到提升（subsection 1.1、subsection 1.2）；
- 已有的多语言内存错误检查工作主要关心多语言互操作的部分情形（在接口层通过 FFI 管理宿主语言对象），仍有多种情形没有得到关注（subsection 1.3）；
- 对于没有关注的情形，在单语言上的相关工作表明，定制需求驱动的指针分析可以提高内存错误检查的精度，同时平衡指针分析的效率；同时多语言指针分析不能简单组合两个单语言指针分析得到（subsection 1.4）。

1.6 指针分析主要工作

指针分析问题提出

[Weihl, 1980] Weihl, W. E. (1980). Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *ACM-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 83–94.

别名分析的出现

[Emami et al., 1994] Emami, M., Ghiya, R., and Hendren, L. J. (1994). Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 242–256.

[Landi and Ryder, 1992] Landi, W. and Ryder, B. G. (1992). A safe approximate algorithm for interprocedural aliasing. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 235–248.

可计算性分析

[Landi, 1992] Landi, W. (1992). Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337.

[Ramalingam, 1994] Ramalingam, G. (1994). The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471.

Andersen/Steensgaard-型指针分析的提出

[Andersen, 1994] Andersen, L. O. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen.

[Steensgaard, 1996] Steensgaard, B. (1996). Points-to analysis in almost linear time. In *ACM-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 32–41.

提高分析精度

上下文（调用点）敏感

[Horwitz, 1997] Horwitz, S. (1997). Precise flow-insensitive may-alias analysis is np-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):1–6.

[Lhoták and Hendren, 2008] Lhoták, O. and Hendren, L. (2008). Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):1–53.

[Reps, 2000] Reps, T. (2000). Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):162–186.

[Shivers, 1991] Shivers, O. (1991). *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University.

[Xu and Rountev, 2008] Xu, G. and Rountev, A. (2008). Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 225–236.

上下文（对象）敏感

[Milanova et al., 2002] Milanova, A., Rountev, A., and Ryder, B. G. (2002). Parameterized object sensitivity for points-to and side-effect analyses for java. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–11.

[Milanova et al., 2005] Milanova, A., Rountev, A., and Ryder, B. G. (2005). Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41.

上下文（类型）敏感

[Smaragdakis et al., 2011] Smaragdakis, Y., Bravenboer, M., and Lhoták, O. (2011). Pick your contexts well: Understanding object-sensitivity. In *ACM-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 17–30.

域敏感

[Heintze and Tardieu, 2001] Heintze, N. and Tardieu, O. (2001). Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*.

流敏感

[Rinetzky et al., 2008] Rinetzky, N., Ramalingam, G., Sagiv, M., and Yahav, E. (2008). On the complexity of partially-flow-sensitive alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):1–28.

提高求解效率

基于 CFL 可达性优化时间效率

[Reps, 1998] Reps, T. (1998). Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726.

[Zhang et al., 2013] Zhang, Q., Lyu, M. R., Yuan, H., and Su, Z. (2013). Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 435–446.

基于约束图优化时间效率

[Hardekopf and Lin, 2007a] Hardekopf, B. and Lin, C. (2007a). The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 290–299.

[Hardekopf and Lin, 2007b] Hardekopf, B. and Lin, C. (2007b). Exploiting pointer and location equivalence to optimize pointer analysis. In *International Static Analysis Symposium (SAS)*, pages 265–280.

[Nasre, 2012] Nasre, R. (2012). Exploiting the structure of the constraint graph for efficient points-to analysis. In *International Symposium on Memory Management (ISMM)*, pages 121–132.

[Rountev and Chandra, 2000] Rountev, A. and Chandra, S. (2000). Off-line variable substitution for scaling points-to analysis. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 47–56.

[Smaragdakis et al., 2013] Smaragdakis, Y., Balatsouras, G., and Kastrinis, G. (2013). Set-based pre-processing for points-to analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 253–270.

基于 BDD 优化空间效率

[Berndl et al., 2003] Berndl, M., Lhoták, O., Qian, F., Hendren, L., and Umanee, N. (2003). Points-to analysis using bdds. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 103–114.

[Bravenboer and Smaragdakis, 2009] Bravenboer, M. and Smaragdakis, Y. (2009). Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 243–262.

[Lhoták, 2006] Lhoták, O. (2006). *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University.

[Whaley et al., 2005] Whaley, J., Avots, D., Carbin, M., and Lam, M. S. (2005). Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 97–118.

[Whaley and Lam, 2004] Whaley, J. and Lam, M. S. (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 131–144.

需求驱动的指针分析

[Bravenboer and Smaragdakis, 2009] Bravenboer, M. and Smaragdakis, Y. (2009). Exception analysis and points-to analysis: Better together. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–12.

[Fu and Ryder, 2007] Fu, C. and Ryder, B. G. (2007). Exception-chain analysis: Revealing exception handling architecture in java server applications. In *International Conference on Software Engineering (ICSE)*, pages 230–239.

[Guyer and Lin, 2003] Guyer, S. Z. and Lin, C. (2003). Client-driven pointer analysis. In *International Static Analysis Symposium (SAS)*, pages 214–236.

[Heintze and Tardieu, 2001] Heintze, N. and Tardieu, O. (2001). Demand-driven pointer analysis. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 24–34.

[Kastrinis and Smaragdakis, 2013] Kastrinis, G. and Smaragdakis, Y. (2013). Efficient and effective handling of exceptions in java points-to analysis. In *International Conference on Compiler Construction (CC)*, pages 41–60.

[Sridharan and Bodík, 2006] Sridharan, M. and Bodík, R. (2006). Refinement-based context-sensitive points-to analysis for java. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 387–400.

[Sridharan et al., 2005] Sridharan, M., Gopan, D., Shan, L., and Bodík, R. (2005). Demand-driven points-to analysis for java. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 59–76.

[Sui and Xue, 2018] Sui, Y. and Xue, J. (2018). Value-flow-based demand-driven pointer analysis for c and c++. *IEEE Transactions on Software Engineering (TSE)*, 46(8):812–835.

[Yan et al., 2018] Yan, H., Sui, Y., Chen, S., and Xue, J. (2018). Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *International Conference on Software Engineering (ICSE)*, pages 327–337.

[Zheng and Rugina, 2008] Zheng, X. and Rugina, R. (2008). Demand-driven alias analysis for c. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 197–208.

精度 VS 效率：可选上下文敏感

[Hassanshahi et al., 2017] Hassanshahi, B., Ramesh, R. K., Krishnan, P., Scholz, B., and Lu, Y. (2017). An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, pages 13–18.

[IBM T.J. Watson Research Center, 2018] IBM T.J. Watson Research Center (2018). Wala: Watson libraries for analysis.

[Jeon et al., 2020] Jeon, M., Lee, M., and Oh, H. (2020). Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–30.

[Jeong et al., 2017] Jeong, S., Jeon, M., Cha, S., and Oh, H. (2017). Data-driven context-sensitivity for points-to analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–28.

[Kastrinis and Smaragdakis, 2013] Kastrinis, G. and Smaragdakis, Y. (2013). Hybrid context-sensitivity for points-to analysis. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 423–434.

[Li et al., 2022] Li, H., Lu, J., Meng, H., Cao, L., Huang, Y., Li, L., and Gao, L. (2022). Generic sensitivity: Customizing context-sensitive pointer analysis for generics. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 1110–1121.

[Li et al., 2018a] Li, Y., Tan, T., Møller, A., and Smaragdakis, Y. (2018a). Precision-guided context sensitivity for pointer analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–29.

[Li et al., 2018b] Li, Y., Tan, T., Møller, A., and Smaragdakis, Y. (2018b). Scalability-first pointer analysis with self-tuning context-sensitivity. In *ACM SIGSOFT Conference on the Foundations of Software Engineering (FSE)*, pages 129–140.

[Li et al., 2020] Li, Y., Tan, T., Møller, A., and Smaragdakis, Y. (2020). A principled approach to selective context sensitivity for pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 42(2):1–40.

[Lu et al., 2021a] Lu, J., He, D., and Xue, J. (2021a). Eagle: Cfl-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–46.

[Lu et al., 2021b] Lu, J., He, D., and Xue, J. (2021b). Selective context-sensitivity for k-cfa with cfl-reachability. In *International Static Analysis Symposium (SAS)*, pages 261–285.

[Lu and Xue, 2019] Lu, J. and Xue, J. (2019). Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–29.

[Oh et al., 2014] Oh, H., Lee, W., Heo, K., Yang, H., and Yi, K. (2014). Selective context-sensitivity guided by impact pre-analysis. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 475–484.

[Oh et al., 2015] Oh, H., Lee, W., Heo, K., Yang, H., and Yi, K. (2015). Selective x-sensitive analysis guided by impact pre-analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(2):1–45.

[Tan et al., 2021] Tan, T., Li, Y., Ma, X., Xu, C., and Smaragdakis, Y. (2021). Making pointer analysis more precise by unleashing the power of selective context sensitivity. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–27.

[Thakur and Nandivada, 2020] Thakur, M. and Nandivada, V. K. (2020). Mix your contexts well: Opportunities unleashed by recent advances in scaling context-sensitivity. In *International Conference on Compiler Construction (CC)*, pages 27–38.

综述

[Hind, 2001] Hind, M. (2001). Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 54–61.

[Kanvar and Khedker, 2016] Kanvar, V. and Khedker, U. P. (2016). Heap abstractions for static analysis. *ACM Computing Surveys (CSUR)*, 49(2):1–47.

[Ryder, 2003] Ryder, B. G. (2003). Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction (CC)*, pages 126–137.

[Smaragdakis et al., 2015] Smaragdakis, Y., Balatsouras, G., et al. (2015). Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69.

[Sridharan et al., 2013] Sridharan, M., Chandra, S., Dolby, J., Fink, S. J., and Yahav, E. (2013). Alias analysis for object-oriented programs. *Lecture Notes in Computer Science*, pages 196–232.

[Tan et al., 2023] Tan, T., Ma, X., Xu, C., Ma, C., and Li, Y. (2023). Survey on java pointer analysi. *Journal of Computer Research and Development*, 60(2):274–293.

Mingzhe Hu
humingzhework@163.com
Hefei, China
March 2023