CertiAllocator

——第一个被自动化验证的页面分配器

韩志磊 谢兴宇

目录

项目背景

形式化验证

- 形式化验证（Formal Verification ）是指使用形式化的逻辑方法证明程序正确性。
- 逻辑系统的三个要素缺一不可：
  - 语法
  - 语义
  - 公式
- 公理系统是可选的，有些验证方法会用到公理（霍尔逻辑），有些则是纯语义的（符号执行）
- 有些验证是可以自动进行的，有些则必须手动完成。一条普遍的规律：自动化程度和表达力成反比。
- 符号执行自动化程度最高，也是表达力最低的方法

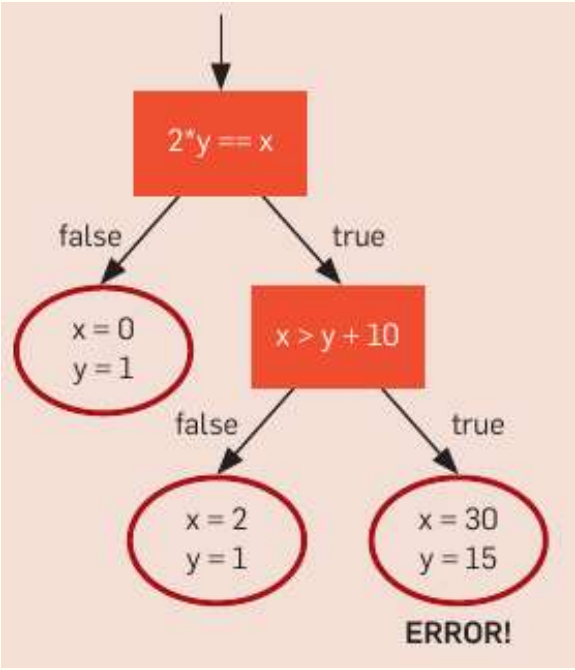
形式化验证

- 有实际意义的程序可以被视作一个函数，规范也是由一个函数表示。
- 函数直接比较是困难的，一般通过外延公理（如果对任意的输入，两个函数的输出都是相同的，则两个函数相等），转化为输入与输出的关系。
- 符号执行的基本思想：向函数输入一系列符号，并模拟函数的执行。
- 因为符号相当于是自由变元，因此外延公理的任意性得以保证。问题被转化为符号执行的结果相等。

符号执行

- 按照程序的语义依次执行。每当执行到分支语句时，执行流会分支，按照分支条件的两种可能分别运行。
- 符号执行引擎为每一个执行实例维护一个path condition(路径条件，简称pc)，代表的是从程序入口执行到当前程序位置所必须满足的条件(也就是每一个分支的执行结果)。

```
void testme (int x, int y) {
          z = twice (y);
          if (z == x) {
                    if (x > y+10)
                              ERROR;
                    }
          }
}
```



符号执行：弊端

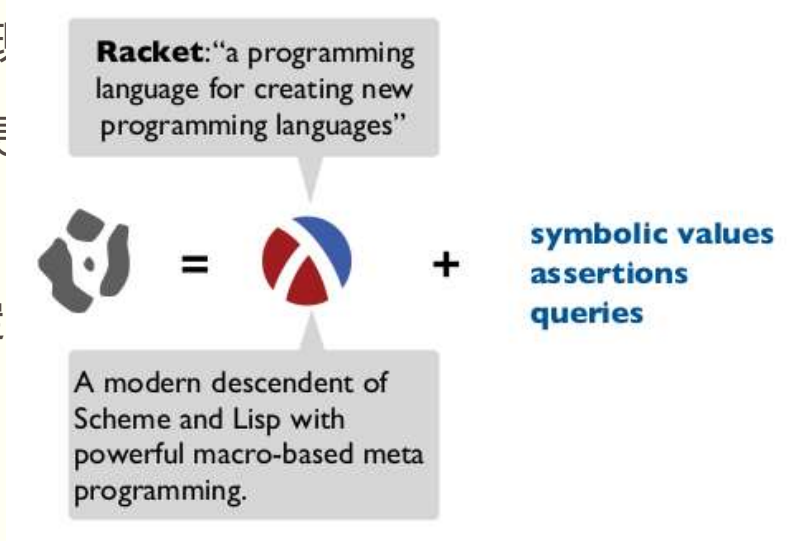
- 语义通过执行得以表现，如果执行次数无法静态确定，符号执行无法停机。具体来说，代码内不能有：
- Unbounded循环
- 递归类型
- Unbounded（互）递归调用

```
void testme_inf() {
        int sum = 0;
        int N = sym_input();
        while (N > 0) {
            sum = sum + N
            N = sym_input();
        }
}
```

- 其次，执行结果的比较需要用约束求解器求解。这限制了程序中的数据类型。以下类型是不被支持的：
- String（字符串函数少部分是可判定的，Rosette中为unsafe）
- 无限精度整型
- 可变容器
- 浮点（很难判定）

验证工具

- Rosette：基于Racket的符号执行引擎
- 其主要优点是可以在同一个语言框架内完成实现和规范的符号执行
  其能力也许不及专用符号执行引擎强大，但是是一种通用符号执行工具。
- Serval：基于Rosette的验证框架
  为验证OS设计，为常用的底层表示都提供了定义。



验证原理

- 我们要验证的是命令式程序，状态是语义的一环。可以认为，状态是函数执行时的上下文。
- 状态通常是庞大且冗余的，大部分的状态信息根本不影响函数的执行结果。
- 抽象是剔除冗余信息的一种方法。抽象函数AF是从实际状态空间S到抽象状态S'的函数。抽象是为了减小符号执行的状态空间
- 问题：如何证明抽象是正确的?

验证原理

- 精化（Refinement）原本指的是对抽象进行调整的过程（CEGAR）。在Serval中，被用来指代对抽象正确性的验证过程。
- $RI(c) \Rightarrow RI(f_{impl}(c))$
- $((RI(c) \wedge AF(c) = s) \Rightarrow AF(f_{impl}(c)) = f_{spec}(s)$
- 可以认为以上二式保证了抽象的正确性
- 精化通过后，就可以在抽象后的状态上进行进一步的验证工作。

验证过程

验证对象：页面分配器

- 我们主要的验证工作集中在ucore的页面分配器，即物理页面的初始化、分配、释放、查询操作。这是一组对OS其他部分暴露的接口，我们验证了其正确性。
- 粗粒度的验证：操作正确地改变了剩余的物理页面数
- 我们的验证是细粒度的，在算法层面验证了ucore的页面分配是first fit的。因此状态空间更大，求解难度也更高。

页面分配器：修改

- 由于符号执行的限制，物理页面数必须是固定的。我们将物理页面改写为了定长数组形式。
- ucore原始的版本使用了链表作为存储物理页面的数据结构。它在实际运行时增删都很高效，但是在符号执行中正好相反，因为指针的存在导致了状态空间膨胀。
- 我们将所有的循环修改为次数为静态可知的，并强制编译器将其展开。
- 其他函数接口也有相应的改变。

```
#if defined(__clang__) && defined(IS_VERIF)
    #pragma clang loop unroll(full)
#endif
    for (size_t p = 0; p < NPAGE; ++p)
        if (base <= p && p < base + n)
            ClearPageAllocated(p);
    nr_free += n;
}
```

页面分配器：验证

- 页面分配器的每一个接口都由一个Rosette编写的函数刻画。
- 验证首先从抽象与精化开始。我们将实际的状态抽象为只包含页面元数据pages以及剩余页面数nr_free的抽象状态。
- 精化是最复杂也最耗时的步骤（必须在实际状态的大状态空间中运行）
- 随后在抽象状态上，为每个具体接口验证了相关的性质：
  - 特征不变式：nr_free==[可分配的页面数量]
  - 不相干性（Noninterference）

安全属性：不相干性（Noninterference）

- 形式化定义（简化自Nickle）

$$\forall tr\hat{}' \in \mathrm{purge}(tr, \mathrm{dom}(a), s)$$
$$\mathrm{output}(\mathrm{run}(\mathrm{s}, tr), a) = \mathrm{output}(\mathrm{run}(\mathrm{s}, tr'), a)$$

- $\mathrm{dom}(a)$：操作a所影响的页面
- 直观含义：对于任意一个操作序列，最后一个操作是a，删掉其中若干（不影响（a影响的页面）的操作）之后，序列的输出不会改变。
- 困难：不相干性几乎无法直接自动证明

Helgi Sigurbjarnarson, Luke NelsonSystems. OSDI'2018. , Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. A Framework for Design and Verification of Information Flow Control

Unwinding Conditions

- (定义) unwinding ($s \overset{u}{\approx} t$): s 和 t 中编号为 u 的页面状态相同
- equivalence relation:
  - reflexivity: $s \overset{u}{\approx} s$
  - symmetry: $s \overset{u}{\approx} t \Rightarrow t \overset{u}{\approx} s$
  - transitivity: $r \overset{u}{\approx} s \wedge s \overset{u}{\approx} t \Rightarrow r \overset{u}{\approx} t$
- local respect: $u \notin \mathrm{dom(a)} \Rightarrow s \overset{u}{\approx} \mathrm{step}(s, a)$
- weak step consistency:

$$s \overset{u}{\approx} t \wedge \left(\forall x \in \mathrm{dom}(a).\, s \overset{x}{\approx} t\right) \Rightarrow \mathrm{step}(s, a) \overset{u}{\approx} \mathrm{step}(t, a)$$

- (参考了 Nickle 的 unwinding conditions)

反思总结

与已有工作的对比

	验证工具	主要验证对象	核心代码行数（实现代码+验证代码）
Ours	Serval	页分配器	170+542
CertiKOS	Coq/Serval	基本进程管理的系统调用	131+421
Komodo	Dafny/Serval	带 enclave 的三级页表	604+1023
A Formally Verified Heap Allocator	Isabelle	堆分配器	未知

Sahebolamri Arash et al. "A Formally Verified Heap Allocator" (2018). Electrical Engineering and Computer Science - Technical Reports.

完整的 ucore 有可能被自动化验证吗？

- 以目前的技术，不可能
- ucore 的编程范式过于复杂，目前的验证技术要求内核必须“验证友好”：
  - 没有循环
  - 没有指针
  - 根据验证的难度随时更改和调整
  - ……
- ucore 的功能过于复杂，对符号执行引擎而言意味着极大的计算量。

```c
// the most complicated syscall in certikos
long sys_spawn(uint64_t fileid, uint64_t quota, uint64_t pid) {
        struct proc *proc;
        uint64_t upper;

        proc = proc_current();

        /* is quota too large? */
        if (proc->upper - proc->lower < quota)
                return -EINVAL;
        /* is pid valid? */
        if (!is_pid_valid(pid))
                return -EINVAL;
        pid = array_index_nospec(pid, NR_PROCS);
        /* does the current process have the permission to allo
        if (!is_pid_owned_by_current(pid))
                return -EINVAL;
        /* has pid been allocated? */
        if (!is_proc_free(pid))
                return -EINVAL;

        /* child takes this new top */
        upper = proc->upper;

        /* take quota off the current process */
        proc->upper -= quota;

        proc_new(pid, proc->next, fileid, proc->upper, upper);

        proc->next = pid;

        return 0;
}
```

```c
// a syscall in ucore
int do_exit(int error_code) {
    if (current == idleproc) {
        panic("idleproc exit.\n");
    }
    if (current == initproc) {
        panic("initproc exit.\n");
    }
    struct mm_struct *mm = current->mm;
    if (mm != NULL) {
        lcr3(boot_cr3);
        if (mm_count_dec(mm) == 0) {
            exit_mmap(mm);
            put_pgdir(mm);
            mm_destroy(mm);
        }
        current->mm = NULL;
    }
    current->state = PROC_ZOMBIE;
    current->exit_code = error_code;
    bool intr_flag;
    struct proc_struct *proc;
    local_intr_save(intr_flag);
    {
        proc = current->parent;
        if (proc->wait_state == WT_CHILD) {
            wakeup_proc(proc);
        }
        while (current->cptr != NULL) {
            proc = current->cptr;
            current->cptr = proc->optr;

            proc->yptr = NULL;
            if ((proc->optr = initproc->cptr) != NULL) {
                initproc->cptr->yptr = proc;
            }
            proc->parent = initproc;
            initproc->cptr = proc;
            if (proc->state == PROC_ZOMBIE) {
                if (initproc->wait_state == WT_CHILD) {
                    wakeup_proc(initproc);
                }
            }
        }
    }
    local_intr_restore(intr_flag);
    schedule();
    panic("do_exit will not return!! %d.\n", current->pid);
}
```

总结：我们做了什么

- 学习自动化验证框架 Serval
- 简化 ucore 中的 page allocator
- 抽象 page allocator 的状态和接口
  - init()
  - init_memmap(base, num)
  - alloc_pages(n)
  - free_pages(base, num)
  - nr_free_pages()
- 验证了两个关键的安全属性
  - nr_free == [the number of available pages]
  - Noninterference
- 首次自动化验证了一个页面分配器

致谢

- 感谢戴臻洋学长的建议和帮助
- 感谢 riscv-ucore 小组提供的 riscv 上的 64 位 ucore
- 感谢陈渝和向勇老师的指导