CertiCore 实验报告

CertiCore 的初衷是做教学操作系统 ucore 的形式化验证版本，但很快便发现这是一个过于宏大的目标。经过半学期的探索，最终实际实现了 ucore 的中断以及页面分配器的验证。但这也并不是一个简单的目标，据我们所知，这是第一个被自动化验证的页面分配器。

背景与验证原理

注：若要读懂此报告的全部细节，要求读者具有一定的数理逻辑、操作系统和符号执行的基础知识。

形式化验证

使用形式化验证的技术对 OS 进行验证已有很长的一段历史。目前主要有三种验证的技术：

1. 交互式定理证明。这种方法使用定理证明器(Proof Assistant)如 Coq 辅助验证，本质上是手动证明，利用证明器的特性可以实现一部分自动化。这种方法要求在形式系统内进行演绎，难度很高。典型的系统包括 seL4 和 CertiKOS。
2. 使用程序标注辅助证明。通过在程序中显式地插入约束、不变量等，利用证明器自动将其转化为约束求解问题进行求解。这种方式的自动化程度更高，但是手动标注依旧很困难。典型的系统如 Komodo。
3. 完全自动化的求解。这种方式期望验证者集中在编写实现接口、规范上，证明器将通过符号执行过程，自动生成约束，问题最终将被转化为可满足性的判定问题。这种方式简单，但是对实现有一些限制，如要求方法必须是有限的。典型的系统是 HyperKernel。

当然，每一种方法都各有优劣。最一般的结论是，验证的自动化程度与表达力基本是成反比的。对于我们在这次实验中使用的符号执行方法而言，其表达力限制就很大。

Xi Wang 等人按照符号执行的思想，开发了Serval框架。此框架实现了状态机精化，符号执行优化等过程，大大简化了系统验证的开销。我们的工作便是在此框架的基础上开展的。

工具：Rosette 与 Serval

Rosette 是基于 Racket 的一门面向验证设计的语言，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。虽然一阶谓词逻辑的可满足性是不可判定的，但是 Rosette 实现的是其中可判定的子集。因此，编写程序规范时，需要一部分技巧来规避表达力不足的问题。

Serval 在此基础上提供了更高级的特性。使用 Serval 验证系统的基本思路是：

1. 使用 Rosette 编写一个 automated verifier, 将系统的实现代码（如汇编，LLVM IR 等）转换为符号执行
2. 使用 Rosette 编写程序规范

Serval 里已经内置了一些常见的 verifer，比如我们用到的 riscv 和 llvm，相当于已经提供了一组相应的语义，因此我们在验证时很少关注 verifer 内部的问题（只在一开始修改了部分实现）。

Serval 会使用 Rosette 生成 SMT 约束，并在此过程中自动进行一些性能的优化。验证者无需关心验证的过程，只需专注在程序接口和规范的编写。

Serval 验证 OS 的基本运行流程是：

1. 将 OS 的 c 和汇编代码编译成汇编代码
    1. 对汇编代码作验证
    2. 可以将得到的汇编代码作为 OS 来运行
2. 将 OS 的 c 和汇编代码编译成 LLVM IR
    1. 对 LLVM IR 作验证

这里分为两部分验证是由于汇编代码的级别太低，不易于直接验证；但在 OS 中，汇编又是无法舍去的部分。所以比较底层的一些属性就直接在汇编代码上来验证，比较高层的一些属性就在 LLVM IR 上来验证。

另外，对 Serval 我们进行了一些修改，增加了一些其没有的支持（主要是 riscv verifier）。在文档写作的时候，Serval 已经有了很大的更新，一部分功能已经被上游提供，但是因为和我们目前修改后的接口不同，要运行 CertiCore 的验证部分，最好还是用我们修改的版本

抽象与精化

Serval 作验证的基本思路是，先将 c 实现的具体状态和具体状态转移函数抽象为 Rosette 中的抽象状态和抽象状态转移函数，并验证这个抽象过程的正确性（这被称作“状态机精化”），之后再验证抽象状态和抽象状态转移函数具有某些安全属性。

Serval 提供了准确的状态机精化支持。在 Serval 中，规范由四部份组成，都在 Rosette 中编写：

1. 程序的抽象状态 s
2. 程序的预期运行行 f_{spec}

3. 一个将程序具体状态映射到抽象状态的 abstract function（AF）
4. 程序执行中的不变式 representation invariant （RI）：用于辅助状态机精化。

Serval 通过以下两式来作状态机精化：

$$RI(c) \Rightarrow RI(f_{impl}(c))$$
$$(RI(c) \wedge AF(c) = s) \Rightarrow AF(f_{impl}(c)) = f_{spec}(s)$$

精化通过后，我们就可以为抽象状态验证一些安全属性（safety property）。比如，欲验证不相干性（noninterference），我们只需考虑抽象状态的转移。Serval 提供了一些内置方法刻画常见的程序属性，为了验证不相干性，经典的做法是验证 local respect 和 weak step consistency，Serval 中便提供了这两个属性的验证函数，具体的细节请看下面安全属性一节。

Timer 的验证

中断的验证基于 Serval 已经提供的 riscv verifier。

比如，以下是一个示例中断处理例程：

```
case IRQ_S_TIMER:
    clock_set_next_event();
    if (++ticks % TICK_NUM == 0) {
        print_ticks();
    }
  break;
```

我们验证了其正确性（也就是 `ticks++`，其余调用并不影响OS的正确性）。在抽象状态中，只有一个 `ticks` 变量，和一个让 `ticks` 的大小增加 1 的抽象状态转移函数：

```
(struct state (tick)
  #:transparent
  #:mutable)
(define (intrp-timer st)
  (set-state-tick! st
  (bvadd (state-tick st) (bv 1 64))))
```

页面分配器的验证

这是我们的工作的主要部分，我们验证的模块在 `default_pmm.c` 中，主要包括：

- `default_init_memmap(base, n)`：初始化从 `base` 开始的 `n` 个页面
- `default_alloc_pages(n)`：分配大小为 `n` 的一段连续页面
- `default_free_pages(base, n)`：释放从 `base` 开始的 `n` 个页面
- `default_nr_free_pages()`：获取当前可用的页面数量（`nr_free`）

这里，我们的具体函数实现和抽象转移函数均采用了 First-Fit 算法，也就是说，当有多个大小 `n` 的连续空闲页面时，我们会选择第一个来分配。

简化

ucore 中原本实现的页面分配器在我们所使用的工具链中是难以被验证的，我们必须要对其进行修改，主要包括：

- 删去指针，使用一个预先分配的页面池 `pages`，而不是用动态分配的链表。
- 对于循环，我们将所有的循环都更改为了循环次数静态有界的形式（`for`套`if`的写法），这样，我们就可以让 clang 把循环静态展开，在生成的 LLVM IR 中不再有循环。

简单地说，这些修改都是符号执行的缺陷导致的。符号执行的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：

- 静态无界循环
- 递归类型
- 静态无界（互）递归调用

当然，也是因为 ucore 本身不是为了验证而设计的，我们必须为了验证（特别的，为了符号执行）而对其进行较大的修改。

精化

数组 `pages` 在抽象状态中以一个函数类型来表示，用 lambda 表达式 $g = \lambda x.ite(x = i) v (f x)$ 表示 `pages` 数据的更新。

First-fit 的页面分配算法的抽象转移函数的实现是这一步的主要难点，为此我们使用了一个辅助函数 `find-free-pages`：

```
(define (find-free-pages s num)
  (define (find-free-accumulate lst acc ans)
    (cond
      [(bveq num acc) ans]
      [(null? lst) #f]
      [(page-available? s (car lst))
        (find-free-accumulate
        (cdr lst) (bvadd1 acc)
        (if (bveq acc (bv 0 64)) (car lst) ans))]
      [else (find-free-accumulate (cdr lst) (bv 0 64))]))
  (define indexl (map bv64 (range constant:NPAGE)))
  (find-free-accumulate indexl (bv 0 64) (bv 0 64)))
```

因为是递归函数，精化过程会比较缓慢。

安全属性

安全属性（safety property），即形式化的的设计意图，用于较为精确地刻画什么样的状态和状态转移函数是安全的、符合设计初衷的。但通常来讲，符合设计意图的抽象状态和抽象状态转移函数难以被几条属性而限定，通常我们只能得到“所有符合设计意图的抽象状态和抽象状态转移函数”的一个上近似（over approximation），即我们选取一些重要的符合设计意图的抽象状态和状态转移函数满足的属性，但满足这些属性的抽象状态和状态转移函数也可能并不是符合我们的设计意图的。

这里，我们提出了两条属性来验证。

特征不变式

我们使用了一个全局变量 `nr_free` 来描述空闲页面的数量，我们希望 `nr_free` 可以准确地描述空闲页面的数量，这可以被精确地形式化为 `nr_free` 的特征不变式

$$\text{nr\_free} == \ | \{p \in \text{all\_pages} \mid p \text{ is available}\} |$$

其中，`all_pages` 是数组中所有的页。

不相干性

不相干性（noninterference）是系统验证和安全领域的一个经典属性，最早由 Goguen 和 Meseguer 在 1982 年提出，这里我们使用的不相干性的形式化定义简化自 Nickle：

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

注：这是一个有类型的公式，每一个变元都有一个约束类型；且此式中的所有自由变元均默认被全称量词约束。

对于公式中涉及的变元的解释：

- a 是一个操作（action），即以下四种操作之一：
  - `default_init_memmap(base, n)`
  - `default_alloc_pages(n)`
  - `default_free_pages(base, n)`
  - `default_nr_free_pages()`
- tr 和 tr' 是操作序列（trace）
- $\text{dom}(a)$ 是操作 a 声明会影响的页面，即：
  - $\text{dom}(\texttt{default\_init\_memmap(base, n)}) = \{p \in \text{all\_pages} \mid base \le p < base + n\}$
  - $\text{dom}(\texttt{default\_alloc\_pages(n)}) = \text{all\_pages}$
  - $\text{dom}(\texttt{default\_free\_pages(base, n)}) = \{p \in \text{all\_pages} \mid base \le p < base + n\}$
  - $\text{dom}(\texttt{default\_nr\_free\_pages()}) = \emptyset$
- s 是一个完整的抽象状态
- $\text{purge}(tr, P, s)$ 是在 tr 中删掉若干 $\text{dom}(a) \cap P = \emptyset$ 后得到的所有可能的新序列。
- $\text{run}(s, tr)$ 是从状态 s 开始顺序执行 tr 中的所有操作之后得到的结果状态。
- $\text{output}(s, a)$ 是 s 执行操作 a 之后得到的所有 $\text{dom}(a)$ 中的页的状态。

非形式化地说，不相干性在这里是指：对于一个以 a 为末尾操作的操作序列，我们把所有不与 a 影响同一个页面的操作称为无关操作，删掉若干无关操作之后，a 的输出不会改变。

但直接证明不相干性的定义对于 Serval 这种基于符号执行的自动证明框架而言，是极为困难的。为了解决这个问题，一个直接的想法是我们可以引入部分人工证明，直接给出若干定理，使得：

- 这些定理是方便自动化推导的
- 这些定理的合取可以推出不相干性（作为已知，不再让符号执行引擎计算）

基于这个思路，我们提出了下面三个定理（参考 Nickle，称作 unwinding conditions）：

- 首先定义 unwinding （记作$s \overset{p}{\approx} t$，其中 s 和 t 是两个状态，p 是一个页面）：s 和 t 中页面 p 的状态是相同的；
- unwinding is a equivalence relation
  - reflexivity: $s \overset{p}{\approx} s$
  - symmetry: $s \overset{p}{\approx} t \Rightarrow t \overset{p}{\approx} s$
  - transitivity: $r \overset{p}{\approx} s \wedge s \overset{p}{\approx} t \Rightarrow r \overset{p}{\approx} t$
- local respect: $p \notin \mathrm{dom}(a) \Rightarrow s \overset{p}{\approx} \mathrm{step}(s, a)$
- weak step consistency: $s \overset{p}{\approx} t \wedge (\forall q \in \mathrm{dom}(a). s \overset{q}{\approx} t) \Rightarrow \mathrm{step}(s, a) \overset{p}{\approx} \mathrm{step}(t, a)$

对于公式中变元的补充解释：

- $\mathrm{step}(s, a)$: 对状态 s 做操作 a 之后得到的状态。

直观地对公式作出解释：

- 要求 unwinding 是 equivalence relation 是很自然的要求
- local respect 说的是操作 a 不会影响 $\mathrm{dom}(a)$ 之外的页面
- weak step consistency 说的是在操作 a 中，a 操作的页面不会被 $\mathrm{dom}(a)$ 之外的页面所影响

解释清楚公式的含义和直观并不是一件容易的事情，对于 noninterference 和 unwinding conditions 更加详尽的解释，建议阅读Nickle的论文。

最后，如果你真的着手实现的话，还有一些细节需要考虑：

- `unwinding` 不能简单的是一个重言式
- 符号量是基于一些特定的类型，但这个类型的所有可能的值并不一定都合法。当符号量是在其类型（比如 `bitvector 64`）中取值时，我们的公式是否依然是正确的？是否能忽略非法的取值？

与已有工作的对比

Serval 重写验证了两个已经被验证的内核，分别是 CertiKOS 和 Komodo；另外，我们还找到了一份对堆分配器的验证工作，验证对象上与我们所验证的页分配器是本质基本相同的。将我们的工作与上述三个工作相比较，结果如下表所示：

	验证工具	主要验证对象	核心实现代码行数	核心验证代码行数
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	enclave + 三级页表	604	1023
A Formally Verified Heap Allocator	Isabelle	堆分配器	未知	未知

注：核心实现代码行数是指被验证的代码，不包括诸如初始化 riscv 的 PMP 和 Supervisor 等部分所需的代码；核心验证代码行数是指对于核心实现代码的抽象转移函数和安全属性。

从上表中，我们可以得到以下三个结论：

- 我们验证的工作量已经达到了验证一个"微内核"的工作量。
- 页分配器一般不会被放到被验证的微内核中，而是会将其放在用户态。
- 页分配器（或堆分配器）功能上其实已经算是比较复杂了。

CertiKOS

CertiKOS 是一个原本使用 Coq 验证的并行操作系统，有 6.5k 行 c 和 x86 汇编代码。Serval 选择了其一个较小的原型实现来改写并验证，代码可见于此，共计有 2k 行 c 和 riscv 汇编代码。

具体而言，对于 CertiKOS，Serval 主要验证的是其关于进程管理的 4 个 sysall：

- `get_quota`：获取 quota，也就是一个用户进程有多少可用的物理页面
- `spawn`：类似于 ucore 中的 `fork`，但会把父进程的物理页面分配给子进程
- `yield`：将进程挂起，调度到下一个进程
- `getpid`：获取当前进程的 `pid`

CertiKOS 对于进程管理的设计大致如下是说，每一个进程的上下文、页表地址等关键寄存器均以 64 位整数的形式被保存在内核中，和进程之间的管理最密切相关的是这五个量：

- `state`: `PROC_STATE_FREE` / `PROC_STATE_RUN`
- `owner`: 父进程的 `pid`
- `next`: 下一个进程的 `pid`
- `lower`: 分配给这个进程的物理页面编号的下界
- `upper`: 分配给这个进程的物理页面编号的上界

CertiKOS 预先开好了一个进程池，进程之间根据其在进程池（也就是一个数组）中的位置构成父子关系，进程 `pid` 的父进程是 `(i >= NR_CHILDREN) ? i / NR_CHILDREN : PID_IDLE`。正在运行的进程被放在通过 `next` 模拟的链表中。每个进程会被分配一段连续的物理页面（quota），子进程的 quota 由父进程的 quota 分配而来。

Serval 对于 CertiKOS 主要验证的安全属性也是进程之间的不相干性，当然这里的不相干性的定义和 unwinding conditions 需要依据抽象状态和转移函数的设计而做出相应的调整，故会稍微复杂一些，也会需要用到一些辅助的不变式，但并不涉及新的概念或设计，这里就不详述了。

Komodo

Komodo 是一个原本使用 Dafny 验证的操作系统，有 2.7k 行 Dafny 代码和 0.9k 行 x86 汇编代码用于实现。同样地，Serval 选择了其一个较小的原型实现来改写并验证，代码可见于此，共计有 2.3k 行 c 和 riscv 汇编代码。

具体而言，Komodo 主要被验证的模块是一个带 enclave 的三级页表。Enclave 是 Intel SGX 引入的一个机制，简单地来说，一个 enclave 就是一片私有的内存区域，其只能被这个 enclave 读写，即便是 OS 也不能读写这个 enclave，不同的 enclave 之间的内存区域也是隔离的，这部分被称作 secure pages。当然，在 enclave 之外，我们还有所有 enclave 和 OS 都能读写的 insecure pages。

主要被验证的 syscall 包括：

- `handle_trap`
- `insecure_read`
- `insecure_write`
- `enclave_read`
- `enclave_write`
- `smc_init_addrspace`
- `smc_init_dispatcher`

- `smc_init_l2ptable`
- `smc_init_l3ptable`
- `smc_map_secure`
- `smc_map_insecure`
- `smc_remove`
- `smc_finalise`
- `smc_stop`
- `smc_enter`
- `smc_resume`
- `svc_exit`

Serval 对于 Komodo 主要验证的安全属性便是 enclave 和三级页表的不相干性，当然由于 enclave 机制的引入，不相干性在 Komodo 上的定义会比在 CertiKOS 上更复杂一些，但在概念上依然遵循通用的不相干性定义，这里就不详述了。

我们目前的缺陷

页面数量

在验证时，我们只保留了一个非常小的页面总量：5。这主要是因为随着页面数量的增长，符号执行的状态空间也会指数级增长，现有的计算能力无法承受。

这主要是因为我们需要操作一段连续的页面，页面之间的状态会相互影响，状态总数满足乘法原理，会指数级增长。我们举一个简化的例子来尝试说明这个问题，我们用 0 代表一个页未被分配，1 代表一个页被分配了。假如某一个时刻所有页的状态是 01001，这时如果我想要释放中间的 3 个页，就需要去查看中间 3 个页的状态，会发现有 2 个页已经是空闲的了，这样“释放中间的 3 个页”就是一个非法操作。所以我们必须要知道所有页的状态，才能确定哪种操作是合法的、哪种操作是非法的。

但事实上，仅仅对我们验证的模块而言，其正确性与页面总量是 5 还是 5k 是无关的，所以尽管 5 是一个看起来不大的数字，但它对我们来说也已经可以接受了。

Rosette 的 bug

Serval / Rosette 虽然是形式化验证的工具，但正所谓医者不能自医，形式化验证的工具链自身并未经过形式化验证，也很有可能有 bug。况且学术界的工具往往不像工业界那样有大量的实际测试机会，其可能会潜藏更多的 bug。

我们在这次验证的经历中也对此深有体会。Rosette 本身是可以支持 statically bounded loop 的，但 Serval 对于 loop 的支持似乎并不完善，导致我们必须在 LLVM 的前端来让编译器直接静态展开。Rosette 对于 `cond` 的支持似乎也有些问题，导致我们在必要时需要用语义上相同的 `when` 来改写。`when` 的条件中如果有符号函数的执行结果，Rosette 对于这种有些扭曲的支持似乎也并不是很好。

由于 Rosette 自身的若干 bug 的存在，我们并未成功实现验证“每个操作都符合每条安全属性”。

ucore 的其他模块

CertiCore 的初衷是对 ucore 作形式化验证，但经过一段时间的探索之后，这个目标被认为是过于宏大的。主要有以下两个原因：

- 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实现代码，最好是同时编写验证和被验证的代码
- 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。被验证的微内核一般是非常简单的，一个简单的对于代码行数的比较是，CertiKOS 被验证的核心代码只有 131 行，而 ucore 的总代码量有 13k 行。

对后续工作的建议

如果之后有同学对基于 Serval 作形式化验证 OS 感兴趣，一个比较可行的思路是选择 ucore 中的另一个模块——比如页表、虚拟存储、进程管理或进程调度器——将其重写得验证友好、抽象状态、验证属性。

那么一个问题是，我们要验证什么样的安全属性？对于大多数模块而言，我认为不相干性便是一个很值得验证的安全属性。当然，每一个模块也会有它自己的独特的安全属性。

验证独立性较强的模块会使许多事情变得比较简单，比如页面置换算法。如果是验证一些与其他模块耦合度比较高的模块，比如进程管理，就会有一些麻烦的问题不得不去考虑：能否将其改写得验证友好、同时又不致损伤其与其他模块的交互？或者该如何把它和其他模块之间的耦合方式更改地更加“验证友好”？