Introduction：

固件模拟的核心问题在于：

理解固件期望从硬件环境中获得什么样的响应。

理想固件模拟的解决方案应该要满足如下四个关键条件：

Virtual

不依赖真实硬件，增强扩展性、减少工程量

Interactive

支持外部输入，内部状态反馈，从而支持动态分析

Abstraction-less：

不依赖一些更高层的抽象（操作系统），从而确保扩展性

Automated

自动化扩展和运行

Pretender：

记录固件与硬件环境的真实交互

利用机器学习和模式识别等技术来为构建外设模型

将外设模型与全系统模拟器结合，从而模拟执行固件

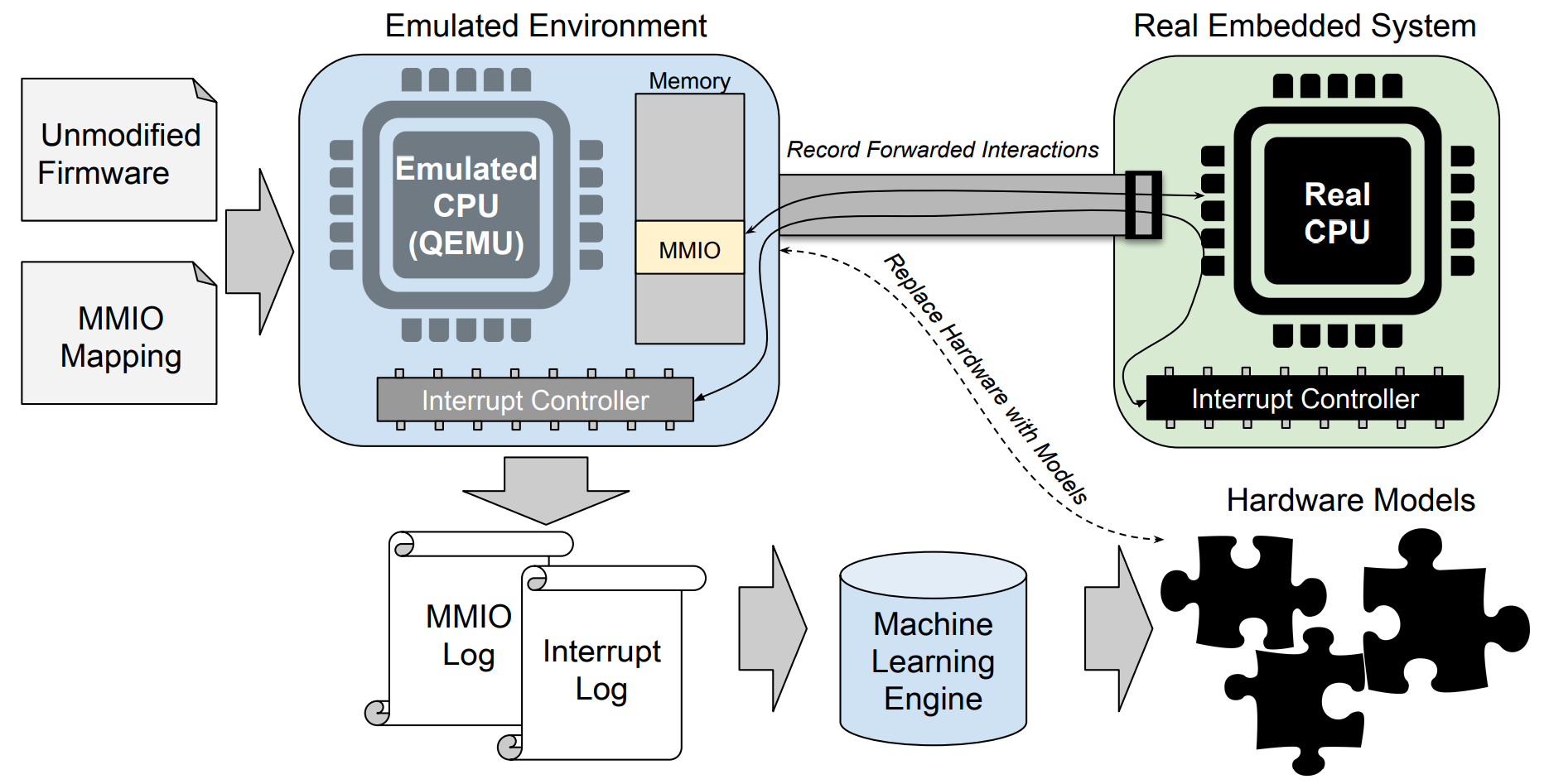
自动化固件模拟的挑战：

外设本身的复杂性（种类繁杂、既是同一个厂商的实现同一种功能的外设，在不同的架构下，其内存布局和实现细节也大不相同）

获得固件与外设的交互信息本身就很具有挑战

按照固件的期望触发中断

Pretender：



前提假设：

1.已经存在CPU和中断控制器的模拟器

2.测试人员能够实时地获取设备固件运行过程中，所有的内存访问和中断触发情况

3.目标设备的基本内存布局可以知悉。（代码、数据、MMIO等区域）

4.能够通过人工或自动化的方式，使固件与硬件充分交互，在Recording phase获得较高的代码覆盖率，使后续的外设建模更加完备。？？？

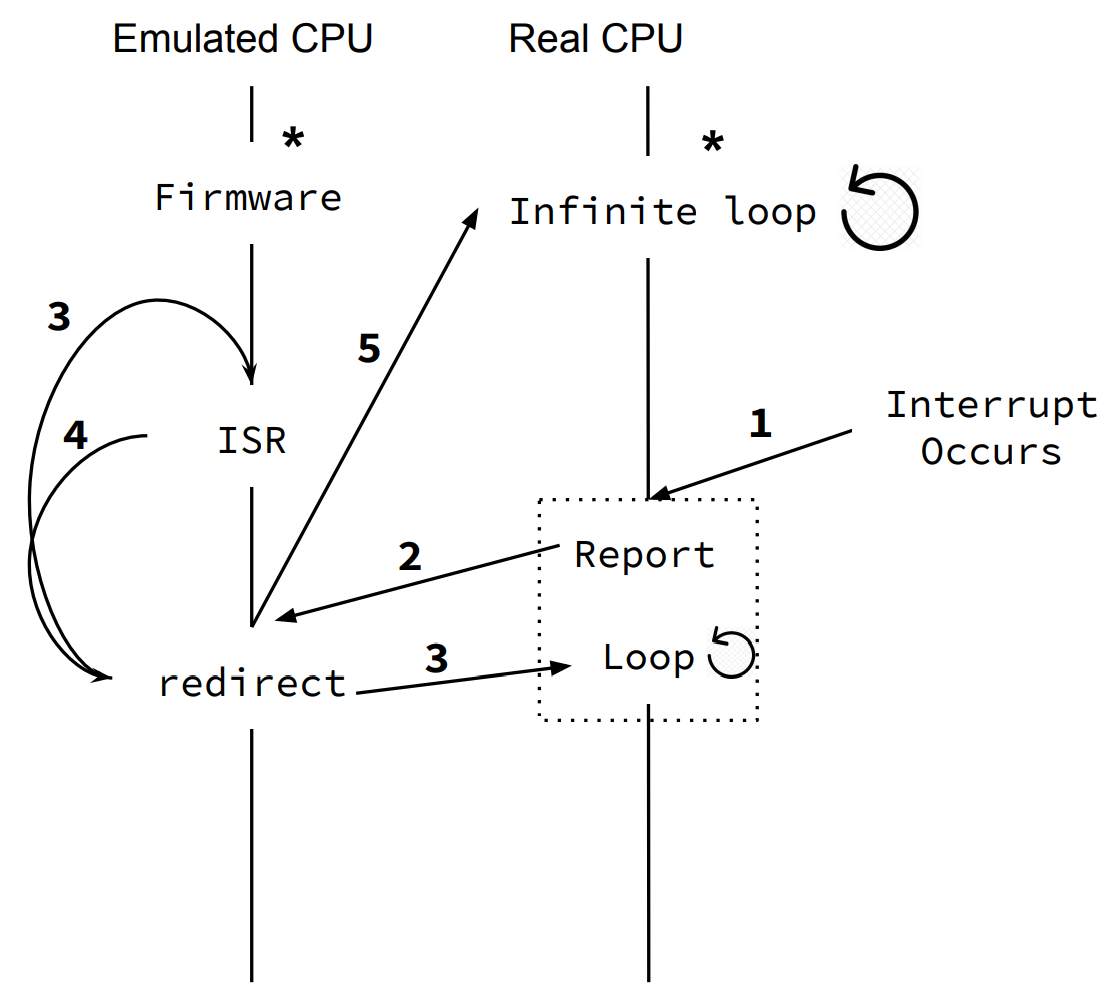
五个阶段：

Recording

记录MMIO访问：

固件与外设的真实交互数据(ARM架构下通过普通的load 和 store指令实现)，通过CPU的内部内存总线流动，无法通过直接截获并记录。

作者利用avatar2框架的hardware-in-loop方式，来实现数据流的记录。在Hardware-in-loop模式下，固件在模拟器中执行，而所有与外设的交互数据，都会被传向真实硬件外设。

记录中断：  


0 设备中，Real CPU跑无限循环，所有的ISR替换为Report

1 外设触发中断

2 设备ISR向Pretender报告中断情况

3 Pretender让Emulated CPU执行相应ISR，同时，Real CPU在中断上下文中运行循环

4 Emulated CPU执行完ISR

5 Real CPU重新执行无限循环

Peripheral Clustering

一个外设存在多个寄存器，对应多个MMIO地址，这些寄存器之间存在联系，因此需要将同一个外设MMIO地址分为一组处理。

作者，通过分析SVD和手动分析真实设备，发现同一个外设对应的MMIO地址区间往往是连续的一小段，并且不同外设的MMIO地址区间存在明显的间隔。

基于上述特性，作者提出使用DBSCAN算法来实现Peripheral Clustering。算法设置，同一个外设的MMIO地址区间最大不超过0x100，最小为1（为了不漏掉简单外设或者不常访问的外设）

Interrupt Inference

一、将中断与外设（MMIO区域）关联

收集ISR期间所有的MMIO操作，从而进行关联

Intuition：

大多数中断的目的在于，让固件与产生中断的外设交互

二、定位中断使能位

在记录中，找到x号中断第一次出现的位置y

继续回溯记录y之前的第一个MMIO write操作的记录z

认为中断使能位就在该MMIO地址 addr指向的内存中

(Intuition：外设配置阶段使能中断的最佳实践）

从记录z往后遍历，分析所有对addr的MMIO write操作

如果某一位被置0，并且之后还触发了中断，那么认为该位不是中断使能位

排除后，剩下的所有位，均认为是中断使能位（尽管实际上，只有1位）

三、确定中断触发策略（频率）

中断有很多种类：

Pulse interrupt

每个事件触发一次

Level interrupt

不断触发，直到一些MMIO操作禁止了中断

仅通过记录，无法区分这两种中断。

最终，作者采用了interrupt timing的策略：

对x号中断，记录每次中断间隔的时间（包括使能中断操作与第一次触发中断的时间间隔），由此构成一个间隔序列。

最终的中断模型：

可以通过使能中断位，来控制中断的开启与关闭；

在固件执行过程中，只要中断被开启，便按照间隔序列触发中断。

Memory Model Training

Simple Storage Model

部分MMIO地址，发挥普通内存的功能。之前写入了什么值，后面读出来就是什么值。

Pattern Model

部分MMIO地址，读出来的值，存在某种明显的模式（包括返回固定值/0 1 1 0 1 1）

Increasing Model

返回值递增的MMIO地址（通常是timer或counter）

其他的模型都较容易实现，但是Increasing Model需要线性回归模型的多轮迭代，来找到最佳拟合。

因为，这些递增寄存器，在boot阶段递增的速率与之后速率不同（固件会在boot后，重新配置速率）

处理方法，经过多轮迭代找到最佳拟合后，将之前的离群值按序保存，Increasing Model会先重放这些离群值（模拟boot阶段），然后再按照线性回归模型返回值。

Write-only Model

在记录中，只被Write的MMIO地址，尽管不需要对其进行额外建模，但是这些地址有可能对应外设的配置寄存器，所以对于外设状态评估有价值。

State Approximation Model

不存在上述模型特点的MMIO地址，一般是片外外设的输入口，反映CPU不可见的复杂状态，并且与中断行为相关。因此，基于函数拟合或状态机直接恢复的方法是不可行的。

Intuition：

MMIO Write操作通常会改变外设的状态

中断同样会改变外设状态

MMIO Read有时也会以隐晦的方式改变外设状态（从串口读取一个字节，会使其从buffer中移除）

Model：

State Approximation Model由对应外设的MMIO和interrupt的trace 以及 state pointer构成。

在模拟执行开始时，state pointer指向trace的开头

在模拟执行过程中，

当Model被Read，从state pointer往后搜索，寻找trace中下一个MMIO Read，返回该Read的Value；如果在找到下一个Read之前，发现了MMIO Write或Interrupt操作，那么就返回最近Value。

（read from MMIO is sequential and respect the boundary of state caused by writes and interrupts）

当Model被Write或相关中断被触发，从state pointer往后搜索，找到Write或Interrupt的记录，更新state pointer到该记录位置。如果搜索到trace末尾仍未找到，那么就从state pointer往回找。如果write的值是全新的（记录中不存在），那么就不更新state pointer。

（作者认为这种方式，能够使得Model更加智能地响应mode和commands中的改变）

Test Harness Creation

利用avatar2的python脚本接口，可以将任意MMIO模型替换为自定义模型，从而实现外部输入。