Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing

如果用一句话来概括Fuzzware这个工作想要做什么，Fuzzware这篇文章是对固件的外设来进行自动化建模以及仿真的工作。

下面我们就来介绍一下为什么要做这个事情，以及具体是怎么做的。

工作背景：为何要对外设进行自动化建模?

由于嵌入式系统计算资源有限，对运行效率敏感，通常并不会如计算机一样。通过牺牲部分性能来部署安全防御手段，此时更容易受到安全漏洞的影响。面对 嵌入式系统软件安全漏洞威胁，开发者需要加强软件测试和质量管理的力度，安全分析者需要不断发掘软件中的安全漏洞威胁，从根源上消除安全漏洞，避免给攻击者留下可乘之机。目前对嵌入式固件进行漏洞挖掘已经成为漏洞挖掘研究的关注热点之一。挖掘嵌入式系统固件中的安全漏洞。其工作思路是借鉴通用计算机上软件漏洞挖掘方案，如模糊测试（Fuzz）等，因此需要具备对固件进行自动化分析的能力。

现有的对固件进行模糊测试的主流方法，有以下四种：

（以下内容引用自From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware）

先介绍对于固件进行fuzzing的四种不同的方式，以及进行对比。

为了检验MCU固件的安全性，已经讨论了几种方法。我们将它们归类为四类。

1. 在真实设备上进行仿真：这种方法需要在真实设备来进行测试。在真实设备上进行测试，它能够得到最真实可信的结果，但可扩展性较差，且缺乏可见性。因为在裸机上进行测试，首先遇到的问题就是收集程序的执行信息较为困难。
2. 完整仿真：为了克服真实硬件上的性能和可扩展性问题，研究人员提出了使用如QEMU这样的完整仿真器来模拟固件执行。主要挑战是仿真不同的外设，并且已经有一些努力朝着这个目标前进。理论上，这种方法能够更好地观察固件执行。不幸的是，据我们所知，目前还没有工作能够精确支持以前未知的设备。因此，目前，这种方法只存在于理想化的设置中。
3. 外设转发：作为一种中间地带的解决方案，混合方法将外设访问转发到真实设备，并将固件在仿真器内运行。然而，由于依赖真实硬件，性能和可扩展性问题仍未解决。

4. 半重新托管：在像HALucinator这样的半重新托管解决方案中，固件的主要逻辑仍然在仿真器内执行。然而，高级HAL函数被识别出来，并用主机上的重新托管处理器替换。因此，避免了对多样化外设的复杂建模。

在真实设备上进行测试的优缺点

对嵌入式固件进行动态测试的最直接方法是测试实体设备，通过网络向其发送数据，进行交互。实体设备是验证漏洞存在、衡量漏洞威胁性的唯一最终标尺，但是依赖实体设备的固件动态分析和漏洞挖掘存在诸多限制，使得效果和效率不如计算机上通用软件的动态测试。
具体来说，有以下几个限制：

1. 设备资源有限，测试效率较低：嵌入式设备经常搭配较低主频的CPU，性能较差，难以满足测试需求。而且单个设备同时只能运行一个测试用例，无法并行进行测试
2. 可调试性不足。内存破坏触发后显示的效果有时并不可见，导致漏洞挖掘效果较差。由于嵌入式固件软件生态封闭，可调试性较计算机软件大大减弱，因此在设备上所能获得的固件运行信息也十分有限。
3. 不可规模化。如果对每种设备的固件测试都需要实体设备，那么对大量固件进行规模化测试将需要花费巨大的人力和物力，因此实体设备不适合规模化 测试。

对固件进行仿真测试的优缺点

固件仿真使得计算机软件上的漏洞挖掘思路可以更容易地迁移到嵌入式固件中， 其中包括模糊测试(Fuzzing)。只要重托管后可以对固件进行插桩，或者对固件的运行状态进行探查，就可以结合 AFL [10]等覆盖率导向的灰盒模糊测试工具，更加 有效地对固件进行充分的软件安全性测试。如果固件仿真环境得到充分有效的实现，其优越性将非常显著。

对固件进行仿真的难点

固件仿真的难点在于：固件与硬件耦合程度高，仿真环境需要为固件提供一定的硬件仿真支持。

以微控制器芯片（单片机）为例，需要仿真的主要组件有以下几个部分：

1. 处理器核心
2. 存储、总线
3. 中断控制器
4. 外设

现有的工具对于主流硬件平台中的 CPU、存储、总线功能仿真支持较好，也可以支持部分类型外设的功能。外设种类众多，并非所有类型的外设都被较好支持。对于不支持模拟的外设，分析者往往需要手动编写代码来仿真外设硬件的行为，或者创建和它们行为等价的模型，才能够使得固件在与外设交互时能够收到正确的外设响应。因此，嵌入式设备外设的模拟自动化成为了嵌入式固件模拟中的一个难点问题，研究者开始探寻有哪些自动化的方案能够摆脱外设模拟能力缺失的困境。本工作就是在嵌入式设备模糊测试领域对外设的自动化建模进行的探索。

Fuzzware的主要工作原理（简介）

Fuzzware是一个纯软件的系统。本工作假设测试者可以获得固件的二进制文件，并对其进行分析。Fuzzware会先对固件进行分析。该工作利用局部作用于的动态执行（DSE）来确定硬件生成的值在固件逻辑中是否有实际意义（例如：确定硬件的值是否有对固件逻辑产生影响）。然后，Fuzzware将分析结果应用与外设模拟过程中，通过这个分析结果来自动生成外设模型，将模糊测试过程集中在变异重要（什么是重要？——这里指的是对固件的逻辑影响较大）的输入上，这极大地提高了其有效性。这篇工作中提到，现有的方法里面会花费大量的时间去变异一个对程序逻辑并没有影响的值。而我们通过对程序在编译阶段的分析，就可以得到哪些值能够有实际的影响。

在对Fuzzware的工作原理进行更加具体的介绍之前，提到了一个路径消除这一基本概念。

Path Elimination

路径消除（Path Elimination）是指在模糊测试的过程中，由于某些原因，某些代码路径没有被执行或测试到，导致这些路径上的潜在漏洞或错误没有被发现和修复。具体来说，路径消除可能发生在以下几种情况：

- 基于引导符号执行的方法：这些方法使用启发式的方法，或者人工预定义的规则来决定哪些路径值得探索，同时在这一过程中可能会丢弃一些路径
  高级仿真：通过抽象和替换固件的部分功能来避免硬件访问，这可能导致某些固件功能未被探索
- 基于模式的MMIO建模：可能错误分类某些寄存器或错误地得出对于给定MMIO访问不存在相关选项的结论，从而可能导致路径消除

路径消除可能会造成的影响有：

- 遗漏固件功能：在执行过程中消除了一些可用的执行路径，导致一些固件功能无法被分析到
- 忽略错误处理和恢复程序：即使在正确建模的情况下，固件可能不会执行到错误处理和恢复逻辑，但这些功能也可能包含漏洞，不应被忽视

因此，路径消除是模糊测试和固件模拟中需要避免的问题，因为它可能导致测试覆盖率不足，从而错过一些重要的安全漏洞。有效的重新托管解决方案应该避免路径消除，同时减少每个固件的手动工作量，

并尽可能消除输入开销。

Fuzzware的主要工作内容（具体）

对MMIO进行自动化建模

MMIO（Memory-Mapped I/O）建模是指在软件仿真环境中对硬件的内存映射输入/输出设备进行模拟的过程。内存映射输入/输出是一种在嵌入式系统和某些计算机体系结构中常用的技术，其中硬件设备（如串口、定时器、网络接口等）的寄存器被映射到处理器的内存空间中。这样，软件可以通过读写这些内存地址来直接与硬件设备通信。

在Fuzzware中，对MMIO进行建模是为了有效地进行固件的模糊测试。由于直接在嵌入式硬件上进行模糊测试通常是不可行的，或者效率非常低，因此Fuzzware采用了重新托管（re-hosting）的方法，即在仿真环境中运行固件。为了使固件能够在这种环境中正常运行，需要对它所依赖的硬件行为进行建模，特别是那些通过MMIO访问的硬件寄存器的行为。

威胁模型

1. 假设能够获得目标设备的二进制固件镜像。
2. 假设提供基本的内存映射，如RAM范围和广泛的MMIO空间
3. 在不了解给定二进制固件映像的特定硬件环境的情况下，我们假设在模糊测试期间攻击者能够控制提供给固件的输入。通常，这些输入可能对应于通过MMIO读取的传入网络数据包的内容、通过串行接口接收的数据或温度测量等传感数据。

Fuzzware采用的关键技术：局部动态符号执行（DSE）

为了分析固件代码的行为，采用动态符号执行（DSE）。DSE允许我们生成一组约束，代表硬件生成值的所有可能用途。评估这些约束使我们能够缩小模糊测试器需要探索的值集。通常，使用符号执行进行建模会因为状态爆炸问题而引入高昂的计算成本。我们通过使用局部DSE来避免这一缺点，其中DSE仅用于执行特定MMIO访问上下文中的代码。

建模方法

1. 对于每个MMIO访问上下文（即程序计数器和被访问的MMIO地址），我们构建一个访问模型。在固件即将执行MMIO访问之前快照模拟器的寄存器和内存状态。在符号执行期间观察到的每个MMIO访问都被视为一个单独的符号变量。在固件执行过程中先确定局部符号执行的分析范围。本工作认为，当发生以下任意情况之一时，就到达了局部符号执行的分析边界：
    i. All tracked symbolic variable are dead (i. e., not alive),
    ii. the current function returns,

iii. a tracked symbolic variable is leaving the scope of theanalysis (i. e., it is written to global memory or to a stack frame of a function higher in the call stack), or
iv. a pre-defined limit of computation resources is exhausted (timeout, number of symbolic states, or number of DSE steps was reached).

2. 模型设计需要保证：
   i. 可复现，多次执行给定原始输入的仿真运行必须产生相同的固件执行结果。
   ii. 保留固件的错误处理路径。

Fuzzware建立的五种MMIO模型

Fuzzware通过分析，对MMIO访问进行了五种模型的建模：

1. 常数模型。这个模型描述了以下情况：MMIO访问了一个常数，这个常数取出来之后会用到判断语句中，下面的程序执行必须满足这个比较才能允许执行继续进行（例如下图中的代码1）。
2. 直通模型。这个模型描述了以下情况：这个模型访问的值硬件生成的值被确定为不影响固件执行逻辑。我们将这种情况下的MMIO访问视为常规内存访问。例如，对配置寄存器的访问（见下图中的代码的2）。
3. 位提取模型。当从MMIO读取的位中只有一部分被固件使用时，使用位提取模型。例如，当从MMIO寄存器读取四个字节并应用掩码仅保留少数位而丢弃其他位时，就是这种情况（见下图中的3）。请注意，对于位移位、截断和等效的指令复合，也会出现类似的效果。
4. 集合模型。集合模型处理的情况是（部分）硬件生成的值被检查以确定不同的值来决定控制流。当可以预计算出一个离散的值列表，其中每个值恰好代表一个可能的控制流选项时，应用此模型。一块原始模糊测试输入被解释为fuzzer在每个单独访问中从不同选项中选择的选项。可能的实例包括状态和识别寄存器，固件根据硬件生成的值执行不同的操作（见图3）。示例：对于一个模型计算出的四个预计算值[1,5,7,128]的2字节宽MMIO访问。模拟器消耗2位模糊测试输入，例如0x1。模拟器提供0x0005作为硬件生成的值。
5. 恒等模型。如果DSE确定硬件生成的值的所有位都是有意义的（即，被固件使用），则分配此模型。它也用作后备方案，以防不受限制的符号变量逃离分析范围，或者如果DSE在其资源限制内未完成。在这些情况下，我们保守地假设硬件生成的值的每个位后来可能被固件使用。因此，我们允许fuzzer尝试所有值，从而发现所有固件路径。正如我们将在第6.1节中展示的，实际上很少需要这种后备方案。

在实际进行模糊测试的过程中，Fuzzware会根据对固件实际执行状态的分析，为每次MMIO访问都匹配这五种预定义模型中的其中一种，然后以此确定采用哪一种模型。

Evaluation

本节提出了以下五个问题：

RQ1 实现的基于符号执行的建模计算成本有多高?
平均下来每生成62个模型需要6.34分钟（也就是说平均每个模型6秒）。
RQ2 由于其保守的定位，FUZZWARE错过了多少优化建模机会?
Fuzzware认为，自己为MMIO分配恒等模型时可以认为是在考虑减少输入开销。在测试中，623次MMIO访问中，只有34个MMIO访问被分配了第五种恒等模型。作者对这34个模型进行了人工验证，其中19个模型的确没有更大的优化空间了，另外15个模型可以进一步缩小模糊测试的探测范围。
RQ3 FUZZWARE的MMIO访问模型是否适用于多种固件和硬件平台?
P2IM提供了一系列测试用例，Fuzzware可以通过所有的测试用例
RQ4 与以前的方法相比，FUZZWARE在模糊测试单体固件方面的表现如何?
Fuzzware与已有工作（μEMU和P2IM）进行对比，发现代码覆盖率和性能方面均优于现有的工作
RQ5 FUZZWARE能否用于发现现实世界固件中以前未知的错误?
报告了15个新的bug和12个新的CVE漏洞。除了显著增加的覆盖率和自动化外，FUZZWARE在三个目标中发现了以前未报告的错误。手动根本原因分析显示，FUZZWARE识别出一个并发问题，一个缺失的指针验证（CNC），以及一个未经检查的AT命令解析崩溃（GPSTracker）。对于所有三个目标，FUZZWARE发现的额外错误与代码覆盖率的显著增加相吻合。

Fuzzware的局限性：

Fuzzware可能会报告这样一些漏洞：实际硬件并不会产生这样的行为，但是建模和实际硬件行为有偏差，导致固件执行了一些在实际情况下根本不会发生的代码路径，从而产生的漏洞。
作者同样提出了这一点，并且作者表示意识到这类问题可能有一些好处：在不同硬件环境中运行的相同固件可能会受到安全漏洞的影响（因为硬件不能被信任）。通过这种方式，FUZZWARE即使在代码部署到不同硬件环境之前，也能指出可能的安全问题。