IoT Security

摘　要

物联网，Internet of Things（IoT），即物与物相连的互联网，是一种在互联网的基础上延伸和扩展的网络。随着智能硬件技术的兴起，物联网发展呈现指数级增长的态势，以美国为首的发达国家将物联网作为国家级战略新兴产业快速推进，我国也将“发展物联网技术和应用”正式列入“十三五”规划之中，物联网已成为技术发展和产业应用的必然趋势。

随着物联网应用数量增加及其规模的扩大，物联网的安全问题引起了企业、消费者和运营商的重视，同时给安全研究人员也带来了挑战。智能家居作为物联网发展应用最广泛的一个领域，它的安全性引发了用户的担忧，由于这些智能设备可以访问非常隐私的数据，用户的个人信息便陷入了被泄露的风险中。

针对以上问题，本文设计并实现了一种基于静态检测的污点分析工具，它可以对物联网应用程序进行检测，来确定该应用程序是否存在泄露用户隐私的风险，并生成所有可能的污点传播路径。而后，本文对该工具的功能进行了改进，包括对隐式流问题的研究以及字符串中敏感词的检测。最后，使用该工具对三星 SmartThings 平台上 180 个官方应用程序和 79 个第三方应用程序进行了评估，并总结评估结果。

关键词：物联网，静态检测，污点分析

IOT SECURITY

ABSTRACT

The Internet of Things (IoT), which means the Internet connected to things, is a network that extends and expands on the Internet. With the rise of intelligent hardware technology, the development of the Internet of Things has shown an exponential growth trend. The developed countries led by the United States have rapidly promoted the Internet of Things as a national strategic emerging industry. China has also officially included the development of Internet of Things technology and applications. In the "13th Five-Year Plan", the Internet of Things has become an inevitable trend of technology development and industrial application.

With the increase in the number of IoT applications and the expansion of their scale, the security of the Internet of Things has attracted the attention of enterprises, consumers and operators, and it has also brought challenges to security researchers. As the most widely used field of Internet of Things development, smart home has caused users' concerns. Because these smart devices can access very private data, the user's personal information is caught in the risk of being leaked.

In view of the above problems, this paper designs and implements a taint detection tool based on static detection, which can detect the Internet of Things application to determine whether the application has the risk of leaking user privacy and generate all possible taint propagation paths. Then, the paper improves the function of the tool, including the study of implicit flow problems and the detection of sensitive words in strings. Finally, the tool was used to evaluate 180 official applications and 79 third-party applications

on the Samsung SmartThings platform and summarize the results.

KEY WORDS: IoT, Static Analysis, Taint Analysis

目 录

插图索引

表格索引

算法索引

第一章　绪论

1.1　物联网与智能家居

物联网，Internet of Things（IoT），即物与物相连的互联网，是一种在互联网的基础上延伸和扩展的网络，它的终端可以是任何智能物品，而并不局限于移动电话、个人电脑等。在物联网上，用户可以将真实的物品通过电子标签连接到互联网，并随时查看它们的活动或状态从而进行一些列的互动。在物联网上，用户可以用控制端（中心计算机）对机器、设备、人员进行集中管理和控制，也可以对家居、汽车设备等进行遥控，亦或者对设备进行定位，防止物品被盗等。物联网也可以起到一种类似于自动化操控系统的作用，它通过收集某些用户个体的数据，从而聚集成大规模的数据库，用来重新设计道路以减少车祸、灾害预测、犯罪防治、流行病控制等社会重大事件，极大方便和改善了人们的生活和工作方式。

随着智能硬件技术的兴起，物联网发展呈现指数级增长的态势，以美国为首的发达国家将物联网作为国家级战略新兴产业快速推进，我国也将“发展物联网技术和应用”正式列入“十三五”规划之中，物联网已成为技术发展和产业应用的必然趋势。

物联网的主要应用包括能源产业和智能电网、联网汽车和运输系统、制造业、可穿戴设备以及植入式设备和医疗设备等。智能家居是物联网发展应用最广的一个领域，它是以住宅为平台，通过物联网技术将家中的各种智能设备连接到一起，旨在构建高效的住宅设施与家庭日程事务的管理系统，提升家居安全性、便利性、舒适性、艺术性，并实现环保节能的居住环境。智能家居设备包括智能锁、智能恒温器、智能开关、智能监控等，它们可以提供家电控制、远程控制、环境监测等各种便捷功能，大大方便了人们的生活和工作。通常，用户使用移动电话作为这些智能设备的控制端，用以管理娱乐、睡眠、烹饪、外出等复杂环境，这也使得对智能家居的控制和操作更加便捷。

1.2　智能家居平台

物联网时代，用户希望可以系统化的管理、使用各种物联网应用程序，同时可以解决兼容性问题，从而将更多智能产品连接起来，最终实现所谓的“万物互联”。于是，智能家居平台的产生很好的满足了这项需求。在智能家居平台上，用户可以下载来自官方或者第三方的应用程序，方便用户对智能设备进行统一的管理。目前主流的智能家居平台包括苹果公司的 HomeKit[1]，谷歌的 Nest[2]，三星的 SmartThings[3]，华为的 HiLink[4] 等。

1.2.1　SmartThings

SmartThings 是三星开发的专有平台。该平台由三个部分组成：HUB、应用程序和云后端。应用程序由 Groovy 语言开发，这是一种动态的，面向对象的语言。该平台上的应用程序称为 SmartApp。

SmartThings 中的权限系统所允许开发人员在安装时指定应用程序所需的设备和用户输入，用以实现应用程序的逻辑，例如通过用户输入设置热水器的温度。此外，SmartThings 中的设备还具有功能（capability），它由动作（action）和事件（event）组成，动作表示如何控制和激活设备，事件则表示设备的状态信息。应用程序通过订阅（subscribe）设备的事件或其他预定义的事件，来调用相应的事件处理程序来执行动作。

对于 SmartThings 平台而言，用户既可以在官方应用程序市场下载 SmartApp，也可以通过其专有的云后端上的 web IDE 安装第三方应用程序。在官方市场上发布的应用程序需要开发人员提交程序的源代码，并通过大约两个月的审查才得以发布，而在 web IDE 上的第三方应用程序不需要任何审核流程，可以直接在 SmartThings 的社区论坛中分析。与其他平台相比，SmartThings 支持更多设备，并且拥有越来越多的官方和第三方应用程序。

1.2.2 OpenHAB

OpenHAB[5] 是一个由 Eclipse IDE 构建的与供应商和技术无关的开源自动化平台。它包括为家庭自动化专门设计的各种设备。OpenHAB 是开源的，提供灵活且可定制的设备集成和应用程序来构建自动化任务。与 SmartThings 平台类似，规则通过三个触发器实现，以响应环境的变化：基于事件的触发器监听来自设备的命令，基于时间的触发器响应特定时间;，基于系统的触发器与某些系统事件（如系统启动和关闭）一起运行。这些应用程序是基于 Xbase 语言的域特定语言（Domain Specific Language）编写的，类似于 Xtend 语言但缺少一些功能。用户可以从 Eclipse IoT marketplace 下载并安装应用程序。

1.2.3 Apple's Homekit

HomeKit 是由苹果公司开发的软件框架，用于管理和控制智能设备。用户可以通过 siri 来语音控制应用程序，从而实现与设备的交互。和 SmartThings 和 OpenHAB 类似，在 HomeKit 中每个设备同样拥有功能，来表示设备可以执行的操作，应用程序通过定义触发器基于位置、设备和时间的事件来执行动作。在 HomeKit 中开发应用程序可以使用 Swift 或 Objective C 编写。用户可以使用 Apple 提供的 Home 移动应用程序安装 HomeKit 应用程序。

1.2.4 IFTTT

IFTTT(If This Then That)[6]，是基于网页服务，将其他应用程序通过简单的条件语句相连接从而生成小程序的平台。严格来说，它不属于物联网平台，而应属于触发动作平台，它通过 API 将第三方的应用程序设置为触发器和反应器，即条件语句的判断条件和执行内容，从而组合成新的小程序。用户不仅可以通过 IFTTT 将非物联网应用程序相关联，也可以使用或自己定制许多将物联网与物联网、物联网与移动端应用程序相关联的小程序。比如，在打开咖啡机后自动发一条微博，或者在打开电灯后让热水器自动进行加热等方便快捷的功能。

1.3 物联网安全

随着物联网应用数量增加及其规模的扩大，物联网的安全问题引起了企业、消费者和运营商的重视，同时给安全研究人员也带来了挑战。对于智能家居而言，在家居设备变得更加智能的同时，用户的隐私也陷入了泄露的风险中。由于这些网络设备可以访问非常隐私的数据，例如智能门锁的密码、电视上观看的内容、以及电灯的开关情况（由此可以判断出家中是否有人或者出门的时间）等，一旦这些信息泄露，将无疑会对用户的财产和人身安全造成巨大损害。

对物联网平台和应用的主要攻击类型包括窃听攻击、伪装欺骗（认证攻击）、访问控制攻击（权限提升）等，其攻击方法也随着时间不断增长。

1.4 静态检测技术

静态检测（Static Analysis）技术是指不运行被测的程序，而仅通过对词法、语法、控制流、数据流等分析来验证程序，通过对代码的自动检测来发现程序所存在的的隐含问题，例如变量未初始化、数据类型不匹配、数组越界、内存泄漏等。静态检测可以帮助程序开发人员自动执行代码分析，快速定位代码的隐藏错误和缺陷，此外静态检测显著减少了在代码上逐行检查的花费的时间，提高软件可靠性并节省了开发和测试的成本。

与静态检测相对应的是动态检测（Dymatic Analysis），是指通过运行被测试的程序，检查运行结果与预期结果之间的差异，并对运行的效率、正确性等性能进行分析的一种技术。

与动态检测相比，静态检测的主要特点包括：

(1) **不实际执行程序**，可以在开发阶段而不是开发完成之后就找出安全漏洞，从而大大降低了漏洞修复的成本。此外，静态检测还可以找到动态检测工具通常无法找到的漏洞，如隐式流问题等。

(2) **执行速度快、效率高**。目前，成熟的静态分析工具可以做到每秒扫描上万行代码，相对于动态检测大幅提高了检测的效率。

(3) **误报率较高**。代码的静态检测通过匹配某种规则来寻找代码中存在的问题，有时可能会将一些正确代码匹配成缺陷问题，这种情况可结合动态检测加以修正。

1.5 污点分析

污点分析是一种跟踪并分析污点信息在程序中流动的技术，其主要概念包括污染源（Taint Source）与污染汇聚点（Taint Sink）。其中，污染源指直接引入不受信任的数据或者隐私数据到程序中，污染汇聚点表示直接产生的安全敏感操作或者泄露隐私数据到外部的操作。污点追踪（Taint Tracking）就是寻找程序中以污染源为起点的数据是否可以直接传播到污染汇聚点，如果存在可行路径则说明该程序存在危险的数据操作或者隐私数据泄露的安全风险，该路径称为污点传播路径。

寻找污点传播路径的原理是，如果污染源中的污点变量通过表达式赋值给第二个变量，那么该变量则被标记为新的污点变量，这两个变量之间形成了一条污点传播路径，之后再由当

前污点变量通过寻找表达式赋值传递给下一个污点变量，以此类推，直到构成一条从污染源到污染汇的传播路径。

以代码1–1为例，变量 a 被预定义的污点函数 source() 标记为污染源，由于第 4 行的变量 x 的值直接依赖于变量 a，因此变量 x 也被标记为污点变量，第 5 行中，污点变量 x 的值又决定了变量 y 的值，因此变量 y 同样被标记为污点变量，又由于变量 y 可以到达地 6 行中的污染汇聚点（预定义的污染汇聚点函数 sink），因此形成了一条由污染源到污染汇聚点的路径: source()->a->x->-y>sink()，该程序存在数据泄露的问题。

代码 1–1 显式流污点传播

```
1 void f(){
2     int a = source()
3     int x,y
4     x = a * 2
5     y = x + 4
6     sink(y)
7 }
```

代码1–1中的污点传播路径是显而易见的，可称之为显式流（explicit flow），与之对应的还有隐式流（implicit flow），如代码1–2中，x 为从污染源得到的污点变量，而 y 与 x 值之间没有直接的数据依赖关系，但是 x 的值却可以隐式的传递给 y，因为 x 的值决定了 for 循环的循环次数，而循环次数又决定了 y 自身迭代的次数，最终决定 y 的取值。仅通过显式流的污点分析，由于没有直接的赋值表达式，y 不会被标记为污点变量，该程序的信息泄露问题被隐藏。隐式流污点分析一直以来都是一个重要的问题，如果未被正确处理将导致污点分析的结果不准确。由于对隐式流问题处理不当而使本应该被标记的污点变量未被标记，导致对所得到的污点传播路径少于真实数量，这样的问题成为欠污染（under-taint）；相反的，如果过多标记污点变量，导致污点传播路径比实际要多，这样的问题称为过污染（over-taint）。当前对隐式流问题的研究主要是减少欠污染与过污染，本文对隐式流问题的处理将在第三章详细说明。

代码 1–2 隐式流污点传播

```
1 void f(){
2     int x = source()
3     int y = 0
4     for(int i = 0 ; i < x ; i++){
5         y = y + 1
6     }
7     sink(y)
8 }
```

1.6 攻击模型及假设

在本文中，我们假设攻击者向用户提供恶意应用程序，或者由官方或第三方向用户提供应非恶意的应用程序，而该应用程序可以获得用户的隐私信息并将其泄露到其他设备或者互联网。这里不必考虑用户是否授予其相关权限，因为应用程序所使用的权限可能会偏离自身

声明的功能，此外，物联网平台授予的权限也可能用于泄露隐私信息。本文假设攻击者无法绕过物联网平台的安全措施，比如伪装成用户进行欺骗攻击等；同时还假设攻击者也无法利用侧信道进行攻击，比如，对于一个控制灯光的应用程序，如果攻击者通过耗电量或者直接观察来得到灯光的开关信息，不属于该应用程序泄露了用户的隐私信息。

1.7 相关工作

对于物联网安全的研究越来越多，但这些研究大都集中在新型物联网编程平台和物联网设备的安全性上。例如，Fernandes[7] 发现了 SmartThings 家庭应用程序权限控制中的设计缺陷，并揭示了越权的严重后果。关于污点分析的研究大都在移动电话平台上，这些研究旨在解决特定领域的挑战，例如为上下文和对象敏感度设计所需算法。

对于在物联网平台中使用污点分析来识别污染源和污染汇聚点的研究，SainT[8] 在之前曾做过类似的工作，但是其功能的完整性和结果的准确度都有待提高，例如，它没有提出解决隐式流问题的有效办法，也未对字符串中的敏感词汇进行标记。此外，由于 SainT 的源代码并非开源，它在设计和实现中所应用的算法并不明确，在其在线检测网站上[9]，部分应用程序并未能识别出污染源与污染汇聚点，原因可能来自算法上的纰漏。

1.8 研究内容及论文构成

基于以上问题，本文设计并实现了一种基于静态检测的污点分析工具，它实现了 SainT 的功能，并提升了其功能的完整性和准确度，用于对基于 Groovy 语言编写的 SmartThings 物联网平台的应用程序进行检测，来确定其是否存在泄露用户隐私的风险，并生成所有可能泄露隐私的污点传播路径，指明污染源与污染汇聚点。而后，使用该工具对 SmartThings 平台 249 个应用程序进行了风险评估。

本文构成为:

(1) **绪论**：介绍相关背景以及解释部分概念。
(2) **静态检测的实现**：详细介绍该工具的实现方法，包括整体流程和实现细节。
(3) **静态检测的改进**：基于已完成的静态检测工具，提出改进与提升的方法。
(4) **检测与评估**：绍用该工具检测物联网应用的过程，并对检测结果进行分析评估。
(5) **结论**：总结全文内容

1.9 本章小结

本章首先介绍了该课题的背景，对物联网的定义、发展物联网的重大意义以及当前物联网安全的现状进行了说明，介绍了物联网的其中一大应用——智能家居，并对当前智能家居的几大主流平台的构成以及开发环境、功能等进行了介绍。此外，本章解释了本文涉及到的几个重要的基本概念，如污点分析、静态检测等，说明了本文的攻击者模型及假设并介绍了相关工作。最后，本章概括了本文的研究内容以及论文的构成。

第二章　静态检测的实现

2.1　总体设计

本文设计并实现了一种用于污点分析的静态检测工具，它的主要工作流程如下：

(1) 将应用程序的源代码转换为便于分析的中间过程代码
(2) 将中间过程代码转化为抽象语法树并输出其遍历结果
(3) 根据抽象语法树的遍历结果划分基本块，生成中间控制流图
(4) 标记控制流图中的污染源与污染汇聚点，通过后向检测技术寻找污点传播的路径并输出。

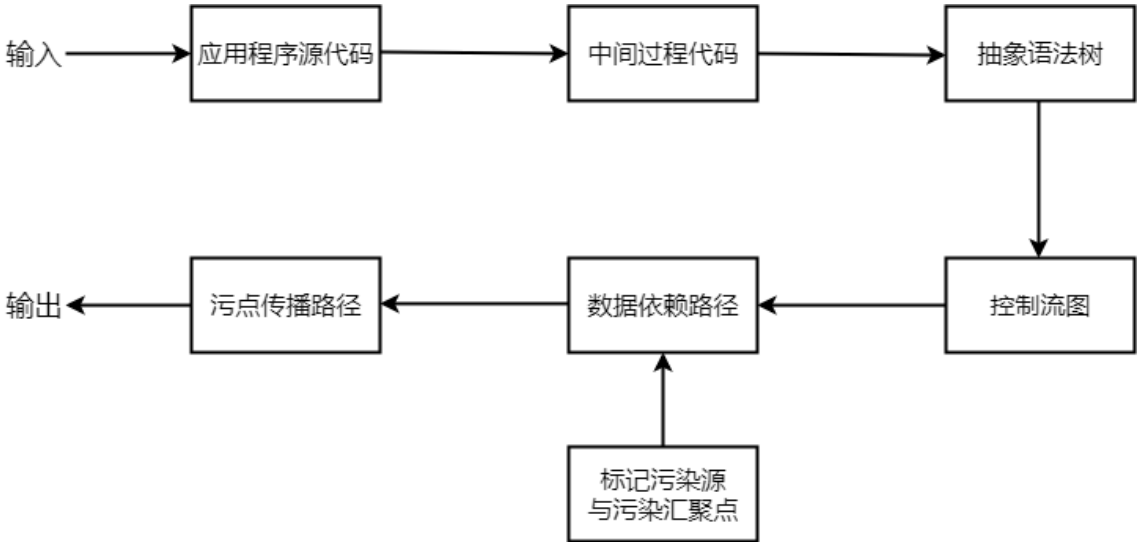

图 2–1 总体设计流程

2.2　标记污染源与污染汇聚点

在本文中，我们将用户的隐私数据标记为污染源中的污点变量，包括以下五个类型的变量：

(1) **用户输入的字符串（text）**：在安装物联网应用程序时，可能会提示用户输入某些字符串，包括个人姓名、手机号、邮箱地址等信息，或者其他用户自定义的字符串，甚至用户输入的账号和密码也可能是遭到泄露。
(2) **用户输入的数值（number）**：在对应用程序进行设定时，可能会输入一些数值，例如设定热水器自动加热的时间和温度，如果用户设定“在 17 点时打开热水器加热至 50 摄氏度”，那么“17 点”与“50 摄氏度”两个变量应当被标记为污点变量。

(3) **设备的信息(device information)**: 包括设备的类型、名字以及它所包含的属性(attribute)等。
(4) **设备的状态（device state)**：例如一个 switch 类型的设备，它当前是处于 on 还是 off，一个 lock 类型的设备正处于 locked 还是 unlock，以及热水器当前的温度等。
(5) **位置信息 (location)**：用户的地理位置信息，包括经度纬度，以及相关的 HUB 信息等。

其中，本文将 (1) 和 (2) 合并为用户输入类，(3) 和 (4) 合并为设备信息类。然后，本文将污染汇聚点划分为两种类型，分别为消息服务和互联网。

(1) **消息服务**：物联网应用程序使用传递消息的 API 向应用程序用户发送推送通知，或者在发生特定事件时向指定收件人发送消息。
(2) **互联网**：物联网应用程序将敏感数据发送到外部服务，如将地理位置发送给某网站获得当地天气；此外，它也可能提供外部用来获取敏感信息的 web 服务，如远程服务器向它请求获取当前的室温值。

最终污点传播示意图如2–2

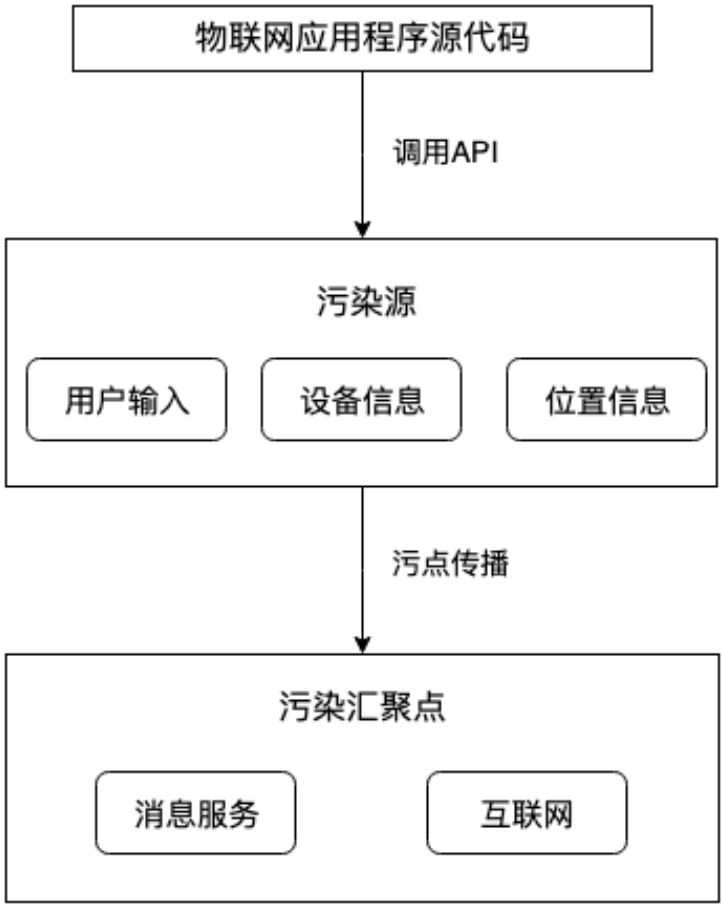

图 2–2 污染源与污染汇聚点

2.3 Groovy 代码

Groovy 是一种基于 java 虚拟机的敏捷的动态语言，结合了 Python、Ruby 以及 Smalltalk 的许多强大功能，Groovy 代码能够与 java 很好的相结合，也能用于扩展现有代码。由于其运行在 java 虚拟机上，groovy 也可以使用 java 语言编写的库。

对于一个由 Groovy 语言编写的物联网应用程序，主要包括 definition、preferences、predefined methods 以及 events handlers 四部分。更多关于 Groovy 的物联网应用程序开发的教程和说明，可见官方教程[10]。

2.3.1 definition

该部分用于描述应用程序的基本信息，包括 App 的名称、开发者、简介以及图标等。比如代码2–1所示，该应用程序的名称是“Good Night”，其描述是：“在夜间特定时间后运动停止时更改模式”。由于该部分不包含程序源码的任何内容，因此可以跳过对该部分的分析。

代码 2–1 definition

```
definition(
    name: "Good Night",
    namespace: "smartthings",
    author: "SmartThings",
    description: "Changes mode when motion ceases after a specific time of night.",
    category: "Mode Magic",
    iconUrl: "https://s3.amazonaws.com/smartapp-icons/ModeMagic/good-night.png",
    iconX2Url: "https://s3.amazonaws.com/smartapp-icons/ModeMagic/good-night@2x.png"
)
```

2.3.2 preferences

该部分主要设置了应用程序的 UI 界面，来引导用户选择设备并为变量赋值。如代码2–2所示，section() 用于表示输入模块，其参数表示该模块的名称，用于提示用户输入。Input 用于用户输入，其中 name 表示输入变量的名称；type 表示其类型，capability.capabilityName 类型表示所有符合该功能的设备；title 是用来提示用户输入的文字。Mutltiple 表示是否允许输入多个变量的值。例如，在名为"When there is no motion on any of these sensors" 的模块中，用户需要在"Where?" 处从所有“motionSensor”中选择需要的“motionSensors”。再比如，在模块"After this time of day" 中，用户需要在 Time? 处输入类型为 time 的名为 timeOfDay 的变量的值。

代码 2–2 preferences

```
preferences {
    section("When there is no motion on any of these sensors") {
        input name:"motionSensors", type:"capability.motionSensor", title: "Where?", multiple: true
    }
    section("After this time of day") {
```

```
        input name:"timeOfDay", type:"time", title: "Time?"
    }
    section("And (optionally) these switches are all off") {
        input name:"switches", type: "capability.switch", multiple: true,
       required:false
    }
}
```

2.3.3 pre-defined methods

由于物联网应用并没有一个 main 函数，所以需要预先定义好一些方法，在该应用程序的周期中可以自动调用，包括四种：

(1) **installed**：当该应用程序被第一次安装时调用。

(2) **updated**：在更新已安装程序的首选项时调用。

(3) **uninstalled**：在卸载该应用程序时调用。

(4) **childUninstalle**：在卸载子应用程序时调用父应用程序。

通常，在所有应用程序中都可以找到 installed() 和 updated() 方法。由于更新应用程序时所选设备可能已更改，因此这两种方法通常都会设置相同的事件订阅（subscribe），通常会将这些调用放入 initialize() 方法并从已安装和更新的方法中调用它。订阅（subscribe）允许应用程序监听设备的事件并调用相应的处理方法，如代码2–3所示，subscribe(p, "presence", h1) 意为，当变量 p 的值为 presence 时，调用事件处理方法 h1()。

通常不需要 uninstalled () 方法，因为系统会在卸载 SmartApp 时自动删除订阅和计划。但是，在与其他系统集成并需要在这些系统上执行清理的应用程序中，它们可能是必需的。

代码 2–3 pre-defined methods

```
def installed() {
    initialize()
}

def updated() {
    unsubscribe()
    initialize()
}

def initialize() {
    subscribe(p, "presence", h1)
}
```

2.3.4 event handlers

Groovy 物联网应用程序的其余部分包含事件订阅中指定的事件处理程序方法以及实现该程序所需的任何其他方法。这些事件处理方法也是该应用程序的入口点。例如在代码2–4中，h1() 就是一个事件处理方法，它根据上一部分订阅的状态变量而执行。x() 是它执行时需要用到的非事件处理方法。

代码 2–4 event handlers

```
def h1(evt){
    x()
}

def x(){
    s.on()
    d.lock()
}
```

2.4 中间过程代码

中间过程代码，即 intermediate representation（IR），是一种源程序的内部表示等效代码，通常包括三元式、四元式、树表示等形式。在本文中，中间过程代码不同于编译原理上的一般概念，而是将 Groovy 的源代码经过简化、提取，得到在表达上更加清晰简洁、易于理解的形式。这样做的好处是，对于那些基于其他任何非 Groovy 语言编写的应用程序，同样可以很容易得到这种表达形式的中间过程代码，作为静态检测的输入从而进行污点分析。同时，中间过程代码删除了许多对污点分析不必要的内容，为下一节中构建控制流图提供方便。

2.4.1 中间过程代码的构成

本文的中间过程代码主要包括三个部分：用户输入、事件订阅以及事件处理。

(1) 用户输入

该部分主要基于 Groovy 源代码的 preferences 模块，通过分析 input() 指令来提取该部分的中间过程代码。对于一个用户输入数值的变量，主要关注三项属性，即变量名称、变量类型以及它是设备还是用户定义的变量，这样做的原因是方便标记污染源的类型。比如，switches 和 lock 是 device 类型，依赖于用户的选择，它们属于设备的信息；而用户定义的 waitfor 直接依赖于用户的输入，属于用户输入的数值类型。

(2) 事件订阅

该部分用以反映事件与动作的对应关系，即当事件被触发时，调动相对应的动作。通过对源代码中 pre-defined method 部分进行分析，寻找 initialize() 或其他预定义的方法中的 subscribe() 指令并加以分析提取，得到包括订阅的变量名称、订阅的事件（变量一旦等于该值）以及该事件发生时所执行的事件处理方法在内的三项内容，有时订阅的事件可以省略，这时只要订阅的变量的值发生更改，就会调用事件处理方法。如程序中，当变量 app 的值发生改变时，事件处理方法 apptouch 就会被触发执行。

这一部分用来衔接用户输入和事件处理，既为程序提示了入口点，也为该入口标记了用户输入的来源。此外，这里的事件不仅限于设备事件，同样适用于 web 服务的事件，由于物联网编程平台允许通过网络访问应用程序，因此外部可以向应用程序发出请求，并获得终端设备的有关信息或控制权限（如 IFTTT）。

(3) 事件处理

为了对污点的传播途径进行分析与追踪，我们需要知道当前应用程序的执行顺序，但是由于物联网应用程序并没有 main 主函数，因此我们需要先找到程序的入口点，再通过调用关系来确定程序的控制流是怎样的，这类似于函数调用图，但区别在于不仅要保留调用关系，还要保留赋值表达式与 If 条件分支表达式，这是因为，调用关系与 If 分支决定了该应用程序的执行顺序结构，赋值表达式影响了污点的传播路径。此外，调用函数如果为 API 则也可能是污染源或污染汇，这些都是污点分析的关键信息。因此，这一部分的中间过程代码将保留源程序中的大量内容。而对于其他一些代码，如 lob.debug（日志信息），对于污点传播或执行顺序没有任何影响，因此可以省略。

最终实现的中间过程代码如2–5所示，其源代码见附录，实际执行过程与中间过程代码的对应关系如图2–3所示。

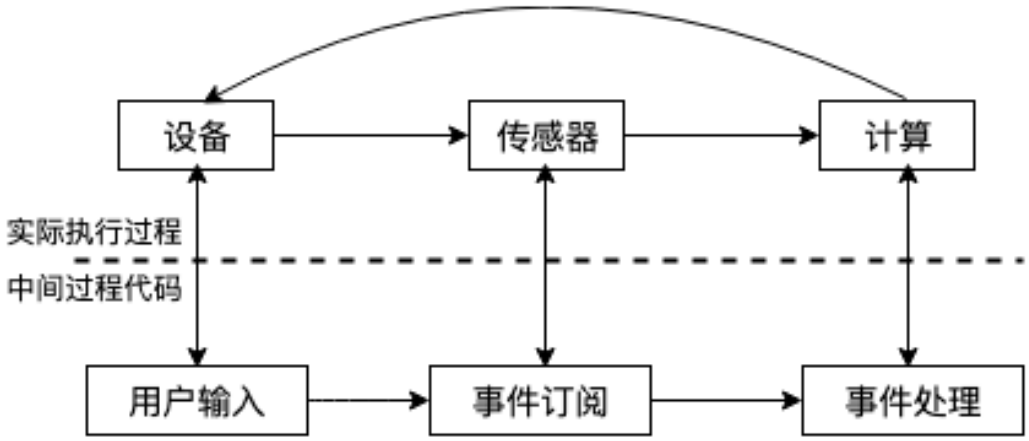

图 2–3 实际执行过程与中间过程代码的对应关系

代码 2–5 中间过程代码示例

```
//用户输入
input (switchesoff, capability.switch, type:device)
input (switcheson, capability.switch, type:device)
input (lock1, capability.lock, type:device)
input (newMode, mode, type:user_defined)
input (waitfor, number, type:user_defined)
//事件订阅
subscribe(app, appTouch)
subscribe(app, appTouch)
//事件处理
def  appTouch( evt) {
    if ( location.mode != newMode ) {
        setLocationMode(newMode)
    }
    lock1.lock()
```

```
16      switcheson.on()
17      def delay
18      if(waitfor != null && waitfor != "")
19          delay = waitfor * 1000
20      else  delay = 120000
21          switchesoff.off(['delay': delay ])
22  }
```

2.4.2 中间过程代码的实现

在本文中，该过程基于抽象语法树（Abstract Syntax Tree）的分析，抽象语法树是源代码语法结构的一种抽象的表现形式，它以树状结构来表现，树上的每一个节点都表示源代码中的一种语法结构，它之所以抽象，是因为真实语法的细节被隐藏在树的结构中，比如，对于表达式赋值，它会将左值、右值以及运算符作为自己的子节点；对于条件分支，它将布尔表达式以及两个条件分支作为自己的子节点；对于函数调用，它则将当前地址、目标函数地址以及需要传递的参数列表作为子节点。

基于抽象语法树，本文使用了 org.codehaus.groovy.ast 中的多个类，并对部分进行了重写（override）。

首先，使用 ASTTransformation 类挂钩（hook）到编译器，这需要先使用 GroovyASTTransformation 类进行注释，指定编译阶段为 SEMANTIC_ANALYSIS。然后，使用 GroovyClassVisitor 提取所要分析的应用程序的入口点和结构，再由 visitMethodCallExpression 提取 AST 节点内的方法调用（包括 preferences 中的 input()，pre-defined methods 中的 subscribe() 以及其他自定义的方法）从而获得我们需要的信息，包括所调用方法的名称以及参数。到此，可以实现中间过程代码的用户输入以及事件订阅部分。

对于事件处理部分，使用 getSourceFromNode 指令将 Method Node 转换为源代码，这样可以保留所有赋值表达式、条件语句和函数调用关系。再通过字符串处理的方法寻找其中不必要的信息并删除。这里可以采用字符串处理的方法，是因为 getSourceFromNode 指令得到的源代码是规范形式，包括自动换行、删除注释、删除多余空格以及补全省略的嵌套括号等。因此，我们可以高效并准确的找到需要删除内容，如 log.debug()，并用括号匹配的方法找到它的参数信息，从而全部进行删除。

用以上方法得到的中间过程代码，对于 Groovy 编译器来说仍然是编译正确的代码，不会产生报错信息，并且保留了源程序的所有重要功能。这也为下一节中由中间过程代码通过 Groovy 抽象语法树得到控制流图提供了用来分析的关键信息。

2.5 控制流图

控制流图，即 Control Flow Graph（CFG），是一个程序的抽象表现形式，它将基本块（Basic Block）作为图的节点，用程序之间可能的执行顺序作为单向的边，从而表示一个程序在执行过程中所有可能会遍历到的路径。

本节中，以中间过程代码的抽象语法树遍历结果作为构建控制流图的输入，在此基础上划分基本块并对其进行连接从而得到控制流图。

2.5.1 抽象语法树的遍历

GroovyASTBrowser 是一种基于 web 的分析 Groovy 程序抽象语法树的工具，需要将源代码输入到 Groovy Console 控制台中运行，从而在新的窗口中得到该应用程序的抽象语法树结构，包括节点之间的从属关系以及各个节点的类型和数值。本文通过修改 GroovyASTBrowser 的源代码，使其可以直接在终端中进行读入与输出。其中，输入为上一节中得到的中间过程代码，输出为该中间过程代码的抽象语法树的深度优先遍历结果，并包括该节点的层级便于进行分析。由于抽象语法树的深度遍历顺序正是程序执行的实际顺序，因此该遍历顺序与源程序代码顺序保持了高度一致，可以用来做进一步的控制流分析。

例如，程序源代码如2–6所示

代码 2–6 程序源代码示例

```
def f()
{
    y=source()
    x=y*3
    if(x>100)
        println x
     else z=g(x)
    sink(z)
 }

 def g(t)
 {
     m=t*10
     return m
 }

```

其得到的抽象语法树遍历结果如2–7所示，其中第一列代表了该节点在抽象语法树中的层级，层级数越大代表该节点距离根节点越远。

代码 2–7 抽象语法树遍历结果

```
3 MethodNode - run
4 BlockStatement - (0)
3 MethodNode - f
4 BlockStatement - (4)
5 ExpressionStatement - BinaryExpression
6 Binary - (y = this.source())
7 Variable - y : java.lang.Object
8 DynamicVariable - y
7 MethodCall - this.source()
8 Variable - this : java.lang.Object
8 Constant - source : java.lang.String
8 ArgumentList - ()
5 ExpressionStatement - BinaryExpression
6 Binary - (x = (y * 3))
7 Variable - x : java.lang.Object
8 DynamicVariable - x
```

```
7 Binary - (y * 3)
8 Variable - y : java.lang.Object
9 DynamicVariable - y
8 Constant - 3 : int
5 IfStatement
6 Boolean - (x > 100)
7 Binary - (x > 100)
8 Variable - x : java.lang.Object
9 DynamicVariable - x
8 Constant - 100 : int
6 ExpressionStatement - MethodCallExpression
7 MethodCall - this.println(x)
8 Variable - this : java.lang.Object
8 Constant - println : java.lang.String
8 ArgumentList - (x)
9 Variable - x : java.lang.Object
10 DynamicVariable - x
6 ExpressionStatement - BinaryExpression
7 Binary - (z = this.g(x))
8 Variable - z : java.lang.Object
9 DynamicVariable - z
8 MethodCall - this.g(x)
9 Variable - this : java.lang.Object
9 Constant - g : java.lang.String
9 ArgumentList - (x)
10 Variable - x : java.lang.Object
11 DynamicVariable - x
5 ExpressionStatement - MethodCallExpression
6 MethodCall - this.sink(z)
7 Variable - this : java.lang.Object
7 Constant - sink : java.lang.String
7 ArgumentList - (z)
8 Variable - z : java.lang.Object
9 DynamicVariable - z
3 MethodNode - g
4 Parameter - t
4 BlockStatement - (2)
5 ExpressionStatement - BinaryExpression
6 Binary - (m = (t * 10))
7 Variable - m : java.lang.Object
8 DynamicVariable - m
7 Binary - (t * 10)
8 Variable - t : java.lang.Object
9 Parameter - t
8 Constant - 10 : int
5 ReturnStatement - return m
6 Variable - m : java.lang.Object
7 DynamicVariable - m

```

2.5.2 基本块的划分

控制流图中，一个基本块代表这之间的代码没有任何跳跃，按照直线顺序依次执行。程序之间的跳跃形式主要包括函数调用、If 条件分支以及循环逻辑结构。在显式流中，循环结构是在一个或多个基本块之中形成闭环，对于污点传播而言，这种环状结构是可以忽略的，因为如果存在一条由循环体的首端到该循环体的尾端的污点传播路径，那么该循环体内所有可能的污点都已经被标记，而无需由尾端回到首端再次进行追踪和标记。因此本节中，基本块的划分主要依据函数调用与 If 条件分支语句，而不考虑循环结构（可将循环次数看做是 1）。

传统意义上的控制流图需要将抽象语法树转换为三地址代码等表现形式，进而划分基本块。但是对于抽象语法树而言，其本身已经包含了划分基本块所需要的所有信息，在此基础上，本文提出了一种可以直接从抽象语法树的深度遍历结果划分基本块的方法，其规则如下：

(1) 所有类型为 MethodNode 的节点都为基本块的首行。MethodNode 为源程序里各个方法的最高层级的节点，其子节点包括函数的参数以及语句块等。如代码2–7中的第 1、3 和 51 行。

(2) 类型为 MethodCall 的节点，并且所调用的方法是在源程序中自定义的，那么该节点是基本块的首行。MethodCall 节点代表了调用其他方法，其子节点包括所调用的方法名称及需要传递的参数。如果该方法来自所导入的库或类（Groovy 默认导入了 java 中的所有库），例如 println(), 那么该调用不会影响该应用程序的控制流，因此可以不做考虑。而如果该调用的方法是源程序中自定义声明的，则该调用必会影响程序的控制流，因此需要作为基本块的首行。如2–7的第 38 行调用了自定义的方法 g()，因此需要被标记为基本块首行，而第 9、28、45 行调用的方法（假设方法 source() 和 sink() 来自导入的库）不需要标记。

(3)（2）中 MethodCall 节点的返回节点（如果有）是基本块的首行。如果该调用是此方法中的最后一条指令，则没有用以返回的节点。如2–7中的第 44 行是调用方法 g() 之后程序返回的节点。

(4) 类型为 IfStatement 的节点及其两个分支语句子节点是基本块的首行。对于 IfStatement 类型的节点，固定有三个子节点，分别是 boolean 条件语句，和两个分支语句块（如无 else，则第二个语句块类型为 EmptyStatement），两个分支语句块相互平行独立，可以作为两个基本块。为方便后续基本块的连接，这里将分支语句块的类型由 BlockStatement 改为 IfChild。如代码2–7中第 21 行的 IfStatement 及第 27 行、第 34 行的两个分支语句被标记为基本块的首行。

(5)（4）中两个分支语句块共同返回的节点（如果有）是基本块的首行。在 If 条件分支结束后，无论实际运行的是哪一个分支语句块，最终都会回到 If 语句的下一条指令。当然，如果在该方法中 If 作为实现该方法的最后一条指令，那么条件分支没有共同的返回节点，无需标记基本块首行。在2–7中，第 44 行的节点是第 21 行 IfStatement 执行完毕后的返回节点，应该被标记。

(6) 所有基本块首行到下一个基本块首行的前一行，构成一个基本块。对每一个基本块依照抽象语法树的遍历顺序进行编号排序，用大写英文字母 A、B、C……为其命名，并记录其首行与尾行的行序号，方便后续分析。

经以上步骤，对2–7的基本块划分结果如2–8所示其中的行数对应2–7中的行数。

代码 2–8 基本块划分结果

```
A,first line:1 ,last line:2
B,first line:3 ,last line:20
C,first line:21 ,last line:26
D,first line:27 ,last line:33
E,first line:34 ,last line:37
F,first line:38 ,last line:43
G,first line:44 ,last line:50
H,first line:51 ,last line:64
```

2.5.3 基本块的连接

基本块作为控制流图的节点，其边是单向连接的，用以表示基本块之间的跳跃顺序，因此需要将基本块按照一定的规则顺序相连。由于各个基本块的首行包含了该基本块的类型信息，因此可以使用如下规则连接基本块：

(1) 首行类型为 IfStatement 的基本块指向它的两个类型为 IfChild 基本块。例如，对于2–8中划分的基本块，应该建立 C → D 和 CE 两条边。

(2) 首行类型为 MethodCall 的基本块指向它所调用的首行类型为 MethodNode 基本块，再由该被调用方法中包含 return 语句的基本块或该方法中的最后一个基本块来指向该 MethodCall 的返回位置。由于在一个方法中，可能含有多个基本块，不能简单的将该方法的第一个基本块，即 MethodNode 基本块指向返回位置，而是应当遍历该方法中所有的基本块，找到包含 return 指令的那一个来指向返回位置，如果该方法没有 return 指令，则应将该方法中的最后一个基本块来指向返回位置。这里所说的返回位置是 2.5.2 节中规则（3）定义的。比如上例中，基本块 F 调用了方法 g()，则建立起 F→H 的边。由于方法 g() 中仅含有 H 一个基本块，所以可以直接由 H 指向返回位置，即建立 H→G 的边。

(3) 首行类型为 IfChild 的基本块需要特别考虑。在一个分支语句块中，可能有多个基本块，比如在分支中包含对其他方法的调用，那么这个分支就会被划分为一条基本块链，因此需要将该链中的最后一个基本块指向返回位置。至于该链中的其他基本块的指向则会由其他规则完成。对两个分支语句块执行同样的操作，完成由条件分支到返回位置的指向。这里所说的返回位置指 2.5.2 节中规则（5）定义的基本块。还需要特殊考虑的是，如果在一个分支基本块链中，它的最后一个基本块是 MethodCall 类型，则不能由它简单指向返回位置，而是由它调用的方法根据规则（2）来指向该条件语句的返回位置。比如在该例中，{D} 和 {E，F} 分别是条件分支的两个基本块链，则 D 可以直接指向它的返回位置 G，即建立 D→G 边；而对于 E 和 F，则需要由 F 指向 G，但又因为 F 是 MethodCall 类型，所以不能构建 F→G 这条边，因此无需操作。

(4) 首行类型为 IfStatement 或 MethodCall 的基本块，由它们的上一个基本块（这里所说的“上一个”是指在基本块编号上相差 1）指向它们自己，除非上一个基本块为 MethodCall

类型，因为调用方法后需要由被调用的方法指向返回位置。在本例中，根据该规则需要建立 B→C，E→F 两条边。

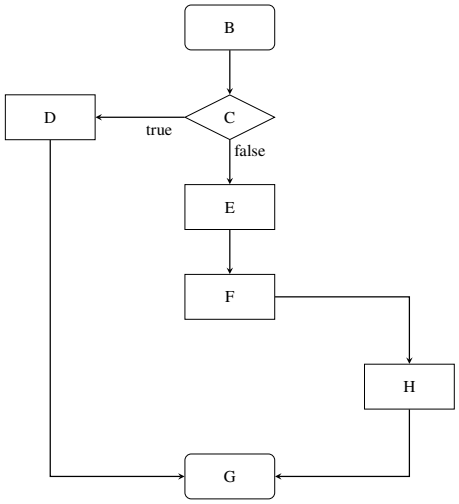

图 2–4 控制流图

最终，得到了程序控制流图的边的集合 {B→C，C→D，C→E，D→G，E→F，F→H，H→G}

通过基本块的划分与连接，得到了中间过程代码的控制流图，如图2–4所示（只考虑中间过程代码中的事件处理部分，用户输入和事件订阅部分由于只包含 input 和 subscribe 两个 API，可不做考虑）。

此外，对于每一个基本块，记录它的入口点和出口点信息，以及它的首行和尾行的行序号，方便下一节中通过控制流图路径对程序进行静态检测。

2.6 后向污点追踪

后向污点追踪（Backward Taint Tracking），即是由污染汇聚点作为起点，由后向前寻找一条到污染源路径的检测方法。使用后向污点追踪方法的原因是，污染汇聚点的数量远远小于污染源，因此从污染汇开始追踪可以大幅度减少开销。

本节将污染汇聚点中所使用的 API 参数标记为敏感信息，沿着程序控制流图向前查找，通过寻找表达式赋值与方法参数调用来确定该敏感信息的数据依赖路径。其具体算法如下：

如算法2–1所示，对污点变量的后向追踪主要由以下几个步骤完成：

(1) 创建名为 worklist，done 和 dep 的三个集合，初始值赋值为空集

(2) 寻找该程序中所有污染汇聚点的 API，并将其控制流图的基本块序号 n 与它的所有参数，即污点变量 id 放入集合 worklist 中。worklist 集合代表待处理的污点变量。例如，对上一节得到的控制流图示例，先将（G，z）放入集合 worklist 中。

(3) 从 worlist 中拿取一个 id 放入 done 中，done 表示已经处理过的污点变量。

算法 2–1 后向污点追踪算法

输入: 中间过程代码的控制流图

输出: 污点变量的依赖路径

1: $worklist \leftarrow 0$
2: $done \leftarrow 0$
3: $dep \leftarrow 0$
4: **for** 污染汇聚点中的每一个参数 id（其所在的控制流图基本块序号为 n） **do**
5: $worklist \leftarrow worklist \cup \{(n,id)\}$
6: **end for**
7: **while** worklist $\neq \emptyset$ **do**
8: $(n,id) \leftarrow worklist.pop()$
9: $done \leftarrow done \cup \{(n,id)\}$
10: **for** 控制流图路径中所有形如 id=f(id*) 的表达式 **do**
11: $ids \leftarrow$ {(n*,id*) | id* 是该表达式中的参数变量，n* 代表其所在的基本块序号}
12: $worklist \leftarrow worklist \cup \{ids \setminus done\}$
13: $dep \leftarrow dep \cup \{n : id, n* : ids\}$
14: **end for**
15: **end while**

(4) 从该 id 的节点 n 沿着控制流图的路径由后向前寻找一个在节点 n* 中的赋值表达式，形如 id=f（id*）,f() 代表对 id* 的某种运算。即 id 的值通过 id* 经过某种运算得到，所以 id* 应当被标记为污点变量。除此之外，如果 id 作为程序内某个方法的参数，那么调用该方法的过程同样应该视作表达式赋值。例如，id 是方法 def g(def id) 的一个参数，那么 def x=g(id*) 同样将 id* 的值赋给了 id。此外，要求从节点 n* 到节点 n 中不存在其他符合条件的赋值表达式。仍然以上节得到的结果为例，z=g(x) 就是一个符合条件的赋值表达式。（为方便说明，这里使用程序源代码，实际操作是基于抽象语法树的遍历结果的）

(5) 将所有符合条件的 id* 放入集合 ids 中，并将集合 ids 里所有不在集合 done 中的元素放入 worklist 中。集合 ids 代表新得到的污点变量，除去已经在集合 done 中完成的污点变量，剩下的这些污点变量是待处理的，所以应该放到集合 worklist 中。即，（F，x）被放入了 worklist 中。

(6) 形成一条由集合 ids 中的所有元素指向 id 的一条路径，将这些路径添加进集合 dep 中。在本例中，得到（G，z）←（F，x）这一条路径。

(7) 重复（2）至（7）的过程，直到集合 worklist 为空集，说明所有污点变量处理完成，已将所有污点传播路径添加到集合 dep 中。

最后，得到（G，z）←（F，x）←（B，y）这样一条数据依赖径。之后检查变量 x 和 y 是否由被标记的污染源 API 得到，如果是则输出这条由污染源到污染汇聚点的污点传播路径。

至此，完成了基于静态检测的污点分析工具的基本功能，对于它的改进将在下一章中介绍。

2.7 本章小结

本节详细阐述了静态检测的实现方法。首先介绍了该静态检测方法的总体设计方案，并对污染源与污染汇聚点进行了标记。然后，对基于 Groovy 语言的物联网应用程序的框架结构进行了介绍，并使用 ASTTransformation 等类提取了应用程序的抽象语法树节点，生成了一种自定义的中间过程代码，从而方便后续的分析并且提高了该方法的可扩展性。之后，由中间过程代码，通过修改 GroovyASTBrowser 工具将其抽象语法树的节点的深度遍历结果输出，根据遍历结果划分基本块并连接，构造出控制流图。最终由控制流图进行后向污点追踪，寻找污染汇聚点的数据依赖路径，并检查污染源，得到污点的传播路径，从而完成静态检测的基本实现。

第三章　静态检测的改进

3.1　隐式流问题

如 1.5 节中所述，污点传播分析中，污点传播不仅依赖于数据，同样还依赖于控制。如果忽略隐式流问题，则有可能出现欠污染的情况；如果对隐式流分析不当，则又有可能导致过污染。本节中同样对隐式流采用静态检测的分析方式，包括对循环结构和条件分支结构两部分中的隐式流问题进行分析。

3.1.1　循环结构中的隐式流问题

如代码3–1所示，在上一章的显式流分析中，污点变量 y 与 x 之间不会产生数据依赖路径，因为并没有变量 x 直接对 y 进行赋值的表达式。但是显然，通过循环结构，y 的值绝对依赖于 x 的值。

代码 3–1 循环结构的隐式流污染示例

```
Void f(){
    Int x=source()
    Int y=0
    For(int i=0;i<x;i++){
        y=y+1
    }
    sink(y)
}
```

在上一章的显式流分析中，污点变量 y 与 x 之间不会产生数据依赖路径，因为并没有变量 x 直接对 y 进行赋值的表达式。但是显然，通过循环结构，y 的值绝对依赖于 x 的值。

一种简单的方法是将循环表达式中的变量全部标记为污点变量，但这样很可能导致过污染问题，因为即使污点变量 y 出现在循环体中，它的值是否依赖于循环表达式还需进一步判断。比如，一个极端的例子3–2，y 是污点变量，但它的值与 x 无关。

代码 3–2 循环结构的隐式流污染特例

```
For(int i=0;i<x;++i)
 {
    y=a
}
```

代码 3–3 循环结构的隐式流污染算法示例

```
For(int i=0;i<x;i++)
{
    a=a+1
    b=i*2
```

```
    y=a
    z=b
}
```

因此，在给循环表达式中的变量标记为污点变量之前，应该先判断循环体中的污点变量的值是否依赖于循环次数。以代码3–3为例，主要分为以下步骤：

(1). 分析循环表达式，提取循环变量 i 和决定循环次数的变量 x，这里包含了四种可能的情况：
   (a). i 的起点值为常数，终点值为常数，如（i=0;i<100;i++)，这种情况可以停止后面的分析，因为污点变量依赖的是常数，不会导污点传播。
   (b). i 的起点值是常数，终点值是变量，如（i=0;i<x;i++)，可以继续后面的分析，因为污点变量的值有可能依赖于 x。
   (c). i 的起点值是变量，终点值是常数，如 (i=x;i<100;++i)，可以继续后面的分析，此情况与上一种相似。
   (d). i 的起点和终点值都是变量，(i=x;i<t;++i)，应当继续分析，而且需要将 x 和 t 都记录下来。

(2). 将循环体中中的全部污点变量标记为 Loop_Tainted，如示例代码3–3，假设之前将 y 和 z 标记为污点变量，则 a、b、i 都应该被标记为 Loop_Tainted。

(3). 检查标记为 Loop_Tainted 的变量是否有对自身赋值的表达式，即左值和右值存在相等的两个变量，如 a=a+1，如果有，则将循环表达式中的所有变量（除 i）标记为污点变量，并建立建立二者之间的数据依赖路径，如 x→a。

(4). 检查标记为 Loop_Tainted 的污点变量是否由循环变量 i 赋值的情况，如 b=i*2，如果有，则，则将循环表达式中的所有变量（除 i 以外）标记为污点变量，并建立建立二者之间的数据依赖路径，如 x→b。

由以上步骤，建立了 x→b→z 和 x→a→y 两条数据依赖路径，完成了对循环结构中隐式流问题的解决。

3.1.2 条件结构中的隐式流问题

除在循环结构外，条件结构中同样存在隐式流问题，而且情况更加多样，涉及的范围更广。如代码3–4：

代码 3–4 条件结构的隐式流污染

```
x=source()
a=''
b=''
if(x==1){
    a='val1'
    b='post'
}
else
    If(x==0){
        a='val2'
        b='post'
```

```
    }
sink(a,b)
```

一种简单的办法，是直接将条件语句中布尔表达式的变量标记为污点变量，并生成它到分支中所有污点变量的数据依赖路径。但是这样可能会导致过污染的情况，如程序所示，污点变量 a 的值依赖于条件分支，但 b 的值在两个分支中相同，因此不能直接生成 x 到 b 的数据依赖路径。

目前而言，还没有通过静态检测对条件结构的隐式流问题完美解决的办法。本文提出一种对条件结构中的污点传播粗略检测的方法。该方法通过寻找并对比污点变量在不同条件分支中的赋值表达式是否相同来决定数据依赖路径。但是对于代码3–5的特殊情况无法检测：

代码 3–5 条件结构的隐式流污染特例

```
x=source()
a=''
if(x==1){
    a='string1'
}
else
    If(x==0){
        a='string2'
        a=a.replace('2', '1')
    }
sink(a)
```

第一个条件分支中 a 的取值为 val1，第二个条件分支中 a 先取值为 val2，但 replace 将 a 中的 2 替换为 1，因此 a 的值变为 val1，与第一个分支相同。这种特殊情况下，污点变量通过不同表达式得到了相同的值，需要通过动态检测查看 a 的具体值，因此本文对此情况不作考虑。

代码 3–6 条件结构的隐式流污染处理方法示例

```
If(x==1){
    a=1
    b=a+3
    c=b*2
    d=0
    e=d+1
    f=d+1
    g=f
}
    Else{
        a=0
        b=a+3
        c=b*2
        d=0
        e=d+1
        f=d+2
        g=f
    }
```

本文以代码3–6为例，说明处理条件结构隐式流问题的方法如下：

(1). 标记两个条件分支中所有污点变量为 if_tainted。
(2). 对于所有 if_tainted 变量，先寻找它的赋值表达式，并判断该表达式的值是否依赖于其他 if_tainted，如果不依赖，则说明它是一个直接通过赋值得到的变量，如 a 和 d。
(3). 对比两个条件分支中 2）中变量的赋值表达式，如果完全相同，如 d，则将该变量标记改为 same_val，如果不同如 a，则将 a 标记为 different_val_leader。
(4). 判断所有 if_tainted 的污点变量，如果它的赋值表达式依赖于一个 different_val，则将它的标记改为 different_val。
(5). 重复步骤 4），直到遍历完所有 if_tainted 变量。如例，abc 都被标记为 different_val，e、f 和 g 仍是 if_tainted。
(6). 对于标记仍然为 if_tainted 变量，需要对比它在两个条件分支中的所有赋值表达式，包括表达式右值的具体内容以及表达式的顺序，如果有不相同则将标记改为 different_val_leader，并执行 4）中步骤，直到遍历完所有剩下的 if_tainted 变量。如，f 被标记为 different_val_leader，g 被标记为 different_val。
(7). 将剩下的 if_tainted 标记改为 same_val，如 f。
(8). 如经过以上步骤，条件结构中所有的污点变量都被标记为 different_val_leader, different_val 和 same_val 三种类型。分别带代表依赖于条件的污点变量和不依赖于条件的污点变量，其中 leader 代表了在这条分支中数据依赖路径的首变量。如果存在 different_val_leader，则将条件结构中布尔表达式中的所有变量标记为污点变量，并形成所有布尔表达式中污点变量到 different_val_leader 的数据依赖路径，如 x→a,x→f。如果全部为 same_val，说明污点变量不依赖于条件表达式，不存在污点传播路径。

以上方法通过对比表达式是否相同来判断污点变量是否依赖于条件表达式，可以解决通过直接赋值的隐式流问题。逐条对比表达式虽然产生了大量的时间开销，但 groovy 物联网应用程序的代码量普遍较少，这种额外的开销对于绝大部分程序可以接受。

3.2 敏感词检测

在上述的静态检测工具中，如果污染汇聚点调用的 api 中参数为字符串，如 SendSms(phone,”The door is locked”)，并没有提出解决办法，因为这种字符串并非来自用户输入，而是官方设定。但是，在字符串中，可能同样包含敏感信息或敏感词汇，比如”Remember to take your Viagra in the cabinet”,“Your gun has been moved”等，这些字符串包含了性、暴力、甚至毒品等词汇，同样泄露了用户的个人隐私。因此本节中使用了 DFA 算法对应用程序中所有的字符串进行敏感词检测，对发现的敏感词用 * 号作处理，如”Remember to take your V*** in the cabinet”, “Your g*** has been moved”.

3.2.1 DFA 算法

DFA(Deterministic Finite Automaton), 即确定有穷自动机，其原理是通过事件 event 由当前状态 state 转到下一个 state，如图3–1。

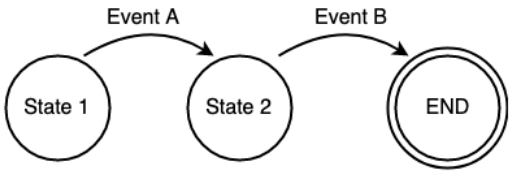

图 3–1 DFA 算法示意

在敏感词检测中，用“查找”来代替 DFA 中的状态和事件。如图3–2，词库中有 drug 和 dress 两个单词，为查找单词 drug，先要查找 d ，得到 r，再查找 r 得到 e 和 u，再查找 u 得到 g，查找 g 得到一个终止节点，代表查找完成。

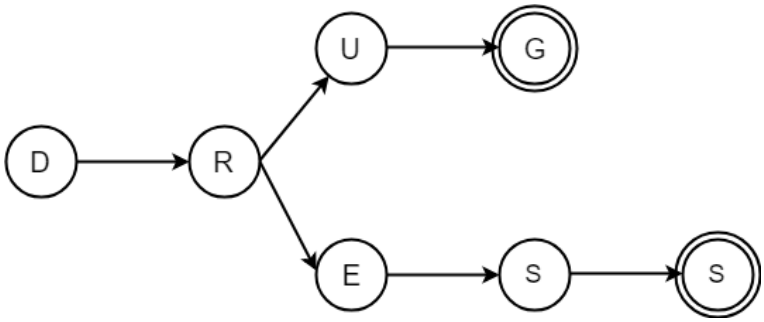

图 3–2 敏感词检测示例

3.2.2 敏感词预处理

将敏感词库中的词进行预处理，得到一个树状结构。对于英文来说每个节点最多有 26 个子节点。创建一个数据结构为树的类，对每一个敏感词使用如下构建方法：

(1). 创建根节点，并设置为当前节点，将首字母设置为当前字母。
(2). 查看当前字母是否是当前节点的子节点，如果不是，则跳至步骤 4，如果是则进行步骤 3。
(3). 将找到的子节点设置为当前节点，并判断该字母是否是改词的最后一个字母（通过当前节点的层级数与原词长度对比来判断），如果是则标记为 END 并退出，否则设置下一个字母为当前字母并重复步骤 2。
(4). 在当前节点中创建该子节点并将其设置为当前节点，并判断该字母是否是终点字母，如果是则标记 END 并退出，否则设置下一个字母为当前字母并重复步骤 2。

通过以上办法，将敏感词库中的所有词加载到了树中，方便接下来的查找。

3.2.3 敏感词的查找

基于上节中得到的敏感词树，对每一个词按照以下办法查找：

(1). 将该词首字母设置为当前字母，根节点为当前节点。

(2). 查找当前节点中是否存在当前字母的子节点，如果不存在则退出，该词不是敏感词，否则执行步骤 3。
(3). 将找到的子节点设置为当前节点，如果该节点被标记为 END 则进行步骤 4，该词是敏感词，否则将下一个字母设置为当前字母，重复步骤 2。
(4). 对该词进行处理，保留首字母，并用 * 号替换后面的字母。

对于一个 Groovy 物联网应用程序，即使最终污染汇聚点中的污点变量是字符串类型，它的值也可能由其他字符串污染而来。因此，可以对源代码中所有类型为字符串的变量统一进行敏感词检测，不必判断它最终是否会出现在污染汇聚点中（因为可能出现条件分支等复杂情况），这样虽然产生了额外的时间开销，但对于普遍较短的物联网应用程序来说是可以接受的，而且这样也确保了可以屏蔽所有敏感词。

3.3 本章小结

本章改进了第二章所设计并实现的静态检测工具。首先，完善了对隐式流问题的研究，分别从循环结构和条件结构进行了说明并提出相应的改进算法。此外，本章还对 Groovy 物联网应用程序中可能泄露用户隐私的字符串问题进行分析，用 DFA 算法预处理敏感词库并查找敏感词，对它进行遮盖处理。通过本章内容，本文设计并实现的静态检测工具功能更加完善、准确率明显提高。

第四章　检测与评估

4.1　检测内容与环境

本文使用上述静态检测工具，对 SmartThings 平台上 249 个应用程序进行了污点分析，包括 180 个官方应用程序和 69 个第三方应用程序。这些应用程序全部来自 IotBench 平台，它主要用来收集物联网应用程序的源代码，方便用户检测。

首先，对于所有应用程序，检测其是否使用了可能导致信息泄露的 API，并标记类型为 Message，Internet 或者二者皆有 Both。其具体 API 名称如表格4–1所示。

表 4–1 污染汇聚点 API

消息服务	互联网
sendSms()	httpDelete()
sendSmsMessage()	httpGet()
sendNotificationEvent()	httpHead()
sendNotificatio()	httpPost()
sendNotificatioToContacts()	httpPostJson()
sendPush()	httpPut()
sendPushMessage()	httpPutJson()

然后，生成污点传播路径，确定污染源的类型，包括用户输入（包括字符串和数值）、设备信息（包括信息和状态）和位置信息三类。其具体 API 名称见附录表格。

最后，分别统计各类数量和比例，绘制统计表。

本文运行环境为 2.20GHz 六核 intel i7 处理器，8GB RAM，IntelliJ IDEA ULTIMATE 2019.1(64bit)，Groovy 2.5.6 版本，Java JDK 1.8.0 版本，平均每个程序的检测运行时间不足 20 秒。

4.2　总体评估

如表格4–2所示，在 249 个 SmartThings 应用程序中，包括 180 个官方应用和 69 个第三方应用。其中，官方应用中存在信息泄露可能的有 89 个占 49.4%，第三方应用中有 48 个，占 69.6%，总计 137 个应用程序有可能泄露用户的信息，占比 55%。

4.3　污染源评估

污染源的共含有用户输入、设备信息和位置信息三个标签。如表格4–3，对于 89 个存在泄露信息可能的官方应用程序，泄露用户输入有 77 个，占 86.5%，泄露设备状态的有 52 个，

占 58.4%，泄露位置信息的有 9 个，占 10.1%。对于 48 个可能泄露信息的第三方应用程序，污染源为用户输入的有 36 个，占 75%，为设备信息的有 24 个，占 50%，为位置信息的有 4 个，占 8.3%。总计，在 137 个可能泄露信息的应用程序中，用户输入共 113 个，占 82.5%；设备信息共 76 个，占 55.5%，位置信息 13 个，占 9.5%。（部分应用程序存在泄露不止一种隐私信息的可能）。

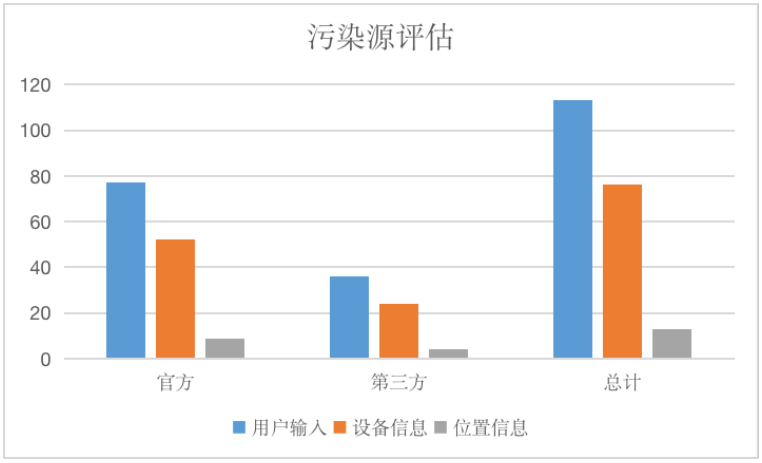

图 4–1 污染源评估

如图4–1所示，用户输入是物联网应用程序所泄露的最多的隐私数据类型。因为大多数物联网应用程序都需要用户输入数据来决定事件触发的条件或者响应措施，而这些输入数据又需要上传到云端或远程服务器，然后再发送给用户。设备信息也是物联网应用程序泄露的最多的隐私数据类型之一，大多数情况是因为云端或者远程服务器需要获得设备的运行状态信息从而进行动作，同时也存在将设备信息发送给用户的情况。只有少部分物联网应用程序需

表 4–2 总体评估

应用类型	总数量	污点应用数量	占比
官方	180	89	49.4%
第三方	69	48	69.6%
总计	249	137	55%

表 4–3 污染源评估

应用类型	总数量	用户输入	设备信息	位置信息
官方	89	77(86.5%)	52(58.4%)	9(10.1%)
第三方	48	36(75%)	24(50%)	4(8.3%)
总计	137	113(82.5%)	76(55.5%)	13(9.5%)

要获得位置信息，泄露的途径主要是上传给 web 用以浏览网页。

4.4 污染汇聚点评估

对于污染汇聚点，分为两种泄露信息的方式，分别为消息服务和互联网。如表4–4中，在官方的 89 个存在泄露隐私风险的应用程序中，污染汇聚点为 message 类型的有 64 个，internet 类型有 18 个，还有 7 个应用程序同时存在这两种污染汇聚点，它们分别占比 71.9%，20.2% 和 7.9%。在 48 个存在泄露风险的第三方应用中，两种类型的污染汇聚点分别有 11 个和 37 个，占 22.9% 和 77.1%，没有同时存在两种污染汇聚点的应用程序。总体而言，75(54.8%) 个应用程序可能通过消息服务泄露用户隐私，55(40.1%) 个应用程序可能从互联网泄露隐私信息，而有 7(5.1%) 个应用程序在这两个方面都存在泄露隐私的风险。

表 4–4 污染汇聚点评估

应用类型	总数量	消息服务	互联网	二者皆有
官方	89	64(71.9%)	18(20.2%)	7(7.9%)
第三方	48	11(22.9%)	37(77.1%)	0
总计	137	75(54.8%)	55(40.1%)	7(5.1%)

4.5 隐式流和敏感词评估

对于隐式流问题，本文分别对是否启用隐式流分析做了两次检测，然而，仅 3 个应用程序只改变了自身的污点传播路径，但并没有影响最终的结果。分析可能的原因，是因为在显式流分析中，污染汇聚点已经泄露了信息。比如 SendSms(phone,msg)，其中 msg 值依赖隐式流，但是 phone 的值由用户直接输入，所以该污染汇聚点已经通过显式流分析确定了结果。

此外，对于敏感词检测，共通过敏感词库检测到了四个涉及药品或暴力的敏感词。对于“Your mail has arrived!”或者“No one has fed the dog!”这种语句，可能同样是用户的隐私信息，但它们不包含敏感词。对此，用户可以自定义敏感词到词库中，防止自己认为是隐私的信息泄露。

4.6 本章小结

该章将本文所设计并实现的工具进行了实际应用，检测了 SmartThings 平台的 249 个应用程序。本章首先介绍了该静态检测工具的检测内容与环境，然后分别从总体、污染源、污染汇聚点、隐式流和敏感词等内容进行了评估，统计了各项应用程序数量及所占比例，并分析数字产生的原因。

第五章　结论

本文设计并实现了一种基于静态检测的污点分析工具，并对该工具的功能进行改进与提升，最终应用到实际中，对 SmartThings 平台 249 个应用程序进行了检测。

本文首先介绍了物联网智能家居、污点传播、静态检测等基本概念，介绍了几个主流物联网平台，并分析物联网智能家居的安全性，在此基础上提出了设计并实现该工具的意义与背景以及相关工作。该工具首先将源代码转换为中间过程代码，再由它的抽象语法树提取控制流图，进行污点追踪。对于隐式流问题，本文对循环结构和条件结构隐式流分别提出了一种检测方法，并添加了敏感词检测功能。最后，本文对 SmartThings 平台进行检测，发现在 249 个 SmartThings 应用程序中，有 55% 的应用程序存在泄露用户隐私数据的风险。随后，分别统计了污染源与污染汇聚点中各污点类型的数量及比例，分析产生的原因，并对该工具自身进行评估。

由于绝大部分物联网应用程序并非开源，本文只对 SmartThings 平台进行了检测，获得的数据较少，日后可通过将其他平台的源代码转换为中间过程代码进行检测；此外，本文对于隐式流分析的效果不明显，可能是因为所检测的物联网应用程序并未包含此等复杂情况，以后应获取更多的检测数据，并应尝试与动态检测相结合。总之，日后将对该工具进行进一步改进，并用来检测更多的物联网平台。

附录 A　代码

A.1　示例应用程序源代码

代码 A–1 中间过程代码示例源代码

```
/**
 *  Good Night House
 *
 *  Copyright 2014 Joseph Charette
 *
 *  Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except
 *  in compliance with the License. You may obtain a copy of the License at:
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software distributed under the License is distributed
 *  on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License
 *  for the specific language governing permissions and limitations under the License.
 *
 */
definition(
    name: "Good Night House",
    namespace: "charette.joseph@gmail.com",
    author: "Joseph Charette",
    description: "Some on, some off with delay for bedtime, Lock The Doors",
    category: "Convenience",
    iconUrl: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience.png",
    iconX2Url: "https://s3.amazonaws.com/smartapp-icons/Convenience/Cat-Convenience@2x.png"
/**
*  Borrowed code from
*  Walk Gentle Into That Good Night
*
*  Author: oneaccttorulethehouse@gmail.com
*  Date: 2014-02-01
 */
 )
preferences {
    section("When I touch the app turn these lights off..."){
        input "switchesoff", "capability.switch", multiple: true, required:true
    }
    section("When I touch the app turn these lights on..."){
        input "switcheson", "capability.switch", multiple: true, required:false
```

```
38      }
39      section("Lock theses locks...") {
40          input "lock1","capability.lock", multiple: true
41      }
42      section("And change to this mode...") {
43          input "newMode", "mode", title: "Mode?"
44      }
45     section("After so many seconds (optional)"){
46          input "waitfor", "number", title: "Off after (default 120)", required:
           true
47      }
48  }
49  def installed()
50  {
51      log.debug "Installed with settings: ${settings}"
52      log.debug "Current mode = ${location.mode}"
53      subscribe(app, appTouch)
54  }
55
56
57  def updated()
58  {
59      log.debug "Updated with settings: ${settings}"
60      log.debug "Current mode = ${location.mode}"
61      unsubscribe()
62      subscribe(app, appTouch)
63  }
64
65  def appTouch(evt) {
66      log.debug "changeMode, location.mode = $location.mode, newMode = $newMode,
          location.modes = $location.modes"
67      if (location.mode != newMode) {
68              setLocationMode(newMode)
69              log.debug "Changed the mode to '${newMode}'"
70      }   else {
71          log.debug "New mode is the same as the old mode, leaving it be"
72          }
73      log.debug "appTouch: $evt"
74      lock1.lock()
75      switcheson.on()
76      def delay
77      if(waitfor != null && waitfor != '' )
78          delay= waitfor * 1000
79      else
80          delay=120000
81      switchesoff.off(delay: delay)
82  }
```

附录 B　表格

B.1　污染源 API

表 B–1 污染源 API（用户输入）

用户输入数值
input"someSwitch","capabllity.switch"
input"someTime","time"
input"someNumber","number"
input"phoneNumber","phone"
用户输入字符串
input"someMessage","text"

表 B–2 污染源 API（位置信息）

位置信息
getContactBookEnabled()
getCurrentMode()
getId()
getHubs()
getLatitude()
getLongitude()
getMode()
setMode()
getTimeZone()
getZipCode()
getLocationId()
getLocation()

表 B–3 污染源 API（设备信息）

设备信息	设备状态
capability.<device type or attribute>	latestState()
getManufacturerName()	statesSince()
getModeName()	getArguments()
getName()	getDateValue()
getSupportedAttributes()	getDescriptionText()
getSupportedCommands()	getDoubleValue()
hasAttribute()	getFloatValue()
hasCapability()	getIntegerValue()
hasCommand()	getJsonValue()
latestValue()	getLastUpdated()
getFirmwareVersionString()	getLongValue()
getId()	getName()
getLocalIP()	getNumberValue()
getLocalSrvPortTCP()	getNumericValue()
getDataType()	getUnit()
getValues()	getValue()
getType()	getData()
getZigbeeId()	getDate()
getZigbeeEui()	getDescription()
events()	getDevice()
eventsBetween()	getDisplayName()
eventsSince()	getDeviceId()
getCapabilities()	getIsoDate()
getDeviceNetworkId()	getSource()
getDisplayName()	getXyzValue()
getHub()	isPhysical()
getLabel()	isStateChange()
getLastActivity()	isDigital()
getManufacturerName()	currentState()
getModelName()	currentValue()
deviceName.capabilities	getStatus()
getTypeName()	

参考文献

[1] Apple's HomeKit[Z]. https://developer.apple.com/homekit/. Accessed May 24, 2019.

[2] Google Nest[Z]. https://nest.com/. Accessed May 24, 2019.

[3] Samsung SmartThings[Z]. https://www.smartthings.com/. Accessed May 24, 2019.

[4] 华为 HiLink[Z]. https://consumer.huawei.com/minisite/smarthome/hilink/index.html. Accessed May 24, 2019.

[5] OpenHAB[Z]. https://www.openhab.org/. Accessed May 24, 2019.

[6] IFTTT[Z]. https://ifttt.com/. Accessed May 24, 2019.

[7] FERNANDES E, JUNG J, PRAKASH A. Security Analysis of Emerging Smart Home Applications[C]//Security & Privacy. [S.l. : s.n.], 2016.

[8] CELIK Z B, BABUN L, SIKDER A K, et al. Sensitive Information Tracking in Commodity IoT[C/OL]//27th USENIX Security Symposium (USENIX Security 18). Baltimore, MD: USENIX Association, 2018: 1687-1704. https://www.usenix.org/conference/usenixsecurity18/presentation/celik.

[9] SainT 在线检测[Z]. https://saint-project.appspot.com/. Accessed May 24, 2019.

[10] SmartThings 开发教程[Z]. https://docs.smartthings.com/en/latest/getting-started/overview.html. Accessed May 24, 2019.

[11] 布莱恩 · 罗素, 德鲁 · 范 · 杜伦. 物联网安全[M]. 北京: 机械工业出版社, 2018.

[12] 阿霍. 编译原理[M]. 北京: 机械工业出版社, 2009.

[13] IoTBench[Z]. https://github.com/IoTBench/IoTBench-test-suite. Accessed May 24, 2019.

[14] Groovy AST Browser[Z]. https://github.com/Beerkay/groovyastbrowser. Accessed May 24, 2019.

[15] 一种基于控制流的污点分析方法[J]. 绵阳师范学院学报, 2018, v.37; No.234(8): 102-106.

[16] 王蕾, 李丰, 李炼, 等. 污点分析技术的原理和实践应用[J]. 软件学报, 2017, 28(04): 120-142.

[17] 污点分析中的隐式污染检测方法[J]. 计算机工程, 2012, 38(23): 28-32.

[18] 张林, 曾庆凯. 软件安全漏洞的静态检测技术[J]. 计算机工程, 2008, 34(12): 157-159.

[19] DFA 算法简介[Z]. https://blog.csdn.net/chenssy/article/details/26961957. Accessed May 24, 2019.

[20] 中英文敏感词库[Z]. https://github.com/fighting41love/funNLP. Accessed May 24, 2019.

[21] Groovy API 2.5.6[Z]. http://docs.groovy-lang.org/docs.. Accessed May 24, 2019.

[22] CELIK Z B, MCDANIEL P D, TAN G. Soteria: Automated IoT Safety and Security Analysis[C]//USENIX Annual Technical Conference. [S.l. : s.n.], 2018.

[23] CELIK Z B, TAN G, MCDANIEL P D. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT[C]//NDSS. [S.l. : s.n.], 2019.

[24] ANDERSEN L O, LEE P. Program analysis and specialization for the c programming language[C]//. [S.l. : s.n.], 2005.

[25] BASTYS I, BALLIU M, SABELFELD A. If This Then What?: Controlling Flows in IoT Apps[C]//ACM Conference on Computer and Communications Security. [S.l. : s.n.], 2018.

[26] ZHANG W, MENG Y, LIU Y, et al. HoMonit: Monitoring Smart Home Apps from Encrypted Traffic[C]//ACM Conference on Computer and Communications Security. [S.l. : s.n.], 2018.

[27] 郭瑶. 浅析中国智能家居发展现状及存在问题[J]. 现代营销 (经营版), 2019(03): 95.

[28] 侯继强, 柴功昊. 智能家居系统在远程命令控制下的数据分递及硬件支持[J]. 无线互联科技, 2019(16): 83-84.

[29] 余洋. “节电管家”手机应用程序交互设计研究[D]. 武汉理工大学, 2016.

[30] 马炜. 基于静态代码分析的日志加强工具的设计与实现[D]. 华中科技大学, 2015.

致　谢

在此，我谨向过去几个月中对我的毕业设计论文提供了帮助的人们表达深深的谢意。

首先，我要向我的导师朱浩瑾教授表示诚挚的感谢。朱浩瑾教授对我的毕业设计进行了精心指导，并且每周检查进度，向我提出改进意见，使我在学习研究、总结学业以及撰写论文等方面有了极大的提高，彰显了朱浩瑾教授高度的敬业精神和责任感。

其次，感谢 NSEC 实验室的同学们，尤其是在读博士生学长张亦弛，指导我研究方向与方法，在课上与课下给予了我很多细节上的帮助，使我的毕业设计得以顺利进行。

最后，特别感谢我的家人，在我遇到困难时给予我鼓励与帮助，让我可以安心完成学业。

四年的本科生涯，虽然辛苦却也收获颇丰，这段生活在老师与同学们的陪伴下即将走向尽头，但我相信这是我即将踏入新征程的开始。值此论文付梓之际，向我四年来所有的教师、同学和家人朋友们表示衷心感谢及真诚的祝福。

IOT SECURITY

The Internet of Things (IoT), which means the Internet connected by things, is a network that extends and expands on the Internet. With the rise of intelligent hardware technology, the development of the Internet of Things has shown an exponential growth trend. The developed countries led by the United States have rapidly promoted the Internet of Things as a national strategic emerging industry. China has also officially included the development of Internet of Things technology and applications. In the "13th Five-Year Plan", the Internet of Things has become an inevitable trend of technology development and industrial application.

In the era of the Internet of Things, users hope to systematically manage and use various IoT applications, and at the same time solve compatibility problems, thereby connecting more intelligent products and ultimately achieving the so-called "Internet of Everything". Therefore, the production of smart home platform meets this demand. On the smart home platform, users can download applications from official or third parties to facilitate unified management of smart devices. The current mainstream smart home platforms include Apple's HomeKit, Google's Nest, Samsung's SmartThings, Huawei's HiLink and more.

With the increase in the number of IoT applications and the expansion of their scale, the security of the Internet of Things has attracted the attention of enterprises, consumers and operators, and it has also brought challenges to security researchers. For smart homes, while home devices become smarter, users' privacy is also at risk of leaking. Since these network devices can access very private data, such as the password of the smart door lock, the content viewed on the TV, and the switch status of the electric light (therefore, it can be judged whether there is someone at home or going out).Once the information is leaked, it will undoubtedly cause great damage to the user's property and personal safety.

In this paper, we assume that an attacker provides a malicious application to a user, or that an official or third-party user provides an application that should be non-malicious, and that application can obtain the user's private information and leak it to other devices or the Internet. . It is not necessary to consider whether the user grants the relevant permissions, because the permissions used by the application may deviate from the functions declared by itself, and the rights granted by the IoT platform may also be used to disclose private information. This paper assumes that an attacker cannot bypass the security measures of the IoT platform, such as disguising as a user for spoofing attacks, and also assumes that an attacker cannot use the side channel to attack. For example, for an application that controls lighting, if an attacker passes Power consumption or direct observation to get the switch information of the light, does not belong to the application reveals the user's private information.

There are more and more researches on IoT security, but most of these studies focus on the security

of new IoT programming platforms and IoT devices. For example, Fernandes discovered design flaws in the SmartThings home application access control and revealed the serious consequences of over-privileged. Most of the research on taint analysis is on the mobile phone platform. These studies aim to solve specific domain challenges, such as designing the required algorithms for context and object sensitivity. SainT has done similar work on the use of taint analysis in the IoT platform to identify sources of taint and taint sink, but its functional integrity and accuracy of results need to be improved, for example, it does not propose an effective way to solve implicitly stream problems, and did not mark sensitive words in strings. In addition, since SainT's source code is not open source, the algorithms it uses in design and implementation are not clear. On its online detection website, some applications fail to identify taint source and taint sink, which may be due to the algorithms.

This paper designs and implements a static detection tool for taint analysis, which is based on (1) converting the source code of the application into intermediate representation for analysis, and (2) converting the intermediate representation into an abstract syntax tree and output the traversal result, (3) according to the traversal result of the abstract syntax tree to divide the basic block, generate the intermediate control flow graph, (4) mark the taint source and taint sink in the flow graph, and search for the taint propagation path through the backward detection and output.

Then, this paper has improved the above static detection tool. In taint analysis, taint propagation depends not only on data, but also on control. If the implicit flow problem is neglected, there may be a situation of under-tainted; if the analysis of the implicit flow is improper, it may lead to over-tainted. In this paper, the implicit detection method of static analysis is used, including the analysis of implicit flow problems both in the cyclic structure and the conditional branch structure, and an algorithm is proposed to deal with such cases.

In the above static detection tool, if the parameter in the API called by the taint sink is a string, such as SendSms(phone, "The door is locked"), no solution is proposed because the string is not from user input. It is the official setting. However, in a string, it may also contain sensitive information or sensitive words, such as "Remember to take your Viagra in the cabinet", "Your gun has been moved", etc. These strings contain words such as sex, violence, and even drugs. It also reveals the user's personal privacy. Therefore, this paper also uses the DFA algorithm to detect sensitive words in all the strings in the application, and treats the sensitive words found with *, such as "Remember to take your V*** in the cabinet", "Your g* ** has been moved".

Finally, this paper uses the above static detection tool to perform taint analysis on 249 applications on the SmartThings platform, including 180 official applications and 69 third-party applications. These applications are all from the IoTBench platform, which is mainly used to collect the source code of IoT applications for user detection. First, for all applications, check to see if they are using APIs that could lead to information disclosure, and mark them as Message, Internet, or both. Then, a taint propagation path is generated to determine the type of taint source, including user input (including strings and values), device information (including information and status), and location information. Finally, the various quantities and proportions are counted separately, and a statistical table is drawn. The

running environment of this paper is 2.20GHz six-core intel i7 processor, 8GB RAM, IntelliJ IDEA ULTIMATE 2019.1 (64bit), Groovy 2.5.6 version, Java JDK 1.8.0 version, the average detection time of each program is less than 20 seconds.

The test found that there are 180 official applications and 69 third-party applications in 249 SmartThings applications, of which 89 are 49.4% of the official applications, and 48 of the third-party applications. 69.6%, a total of 137 applications have the potential to disclose user information, accounting for 55%. For the 89 official applications with leaked information, there were 77 leaked user input, accounting for 86.5%, 52 leaked device status, accounting for 58.4%, and 9 leaked location information, accounting for 10.1%. For the 48 third-party applications that may leak information, the taint source is 36 for the user input, accounting for 75%, 24 for the device information, accounting for 50%, and 4 for the location information, accounting for 8.3%. In total, among the 137 applications that may leak information, a total of 113 users entered, accounting for 82.5%; a total of 76 device information, accounting for 55.5%, and 13 location information, accounting for 9.5%. (Some applications have the potential to leak more than one type of private information). In the official 89 applications with leaked privacy risks, there are 64 taint sink for the message type, 18 for the internet type, and 7 applications with both taint sink, which account for 71.9%, 20.2% and 7.9%. Among the 48 third-party applications with risk of leakage, there are 11 and 37 taint sink, respectively, accounting for 22.9% and 77.1%. There are no applications for both taint sink. For the implicit flow problem, this paper separately tested whether to enable implicit flow analysis. However, only 3 applications changed their own taint propagation path, but did not affect the final result. The possible reason for the analysis is because in the explicit flow analysis, the taint sink has leaked information. For example, sendSms (phone, msg), where the msg value depends on the implicit stream, but the value of the phone is directly input by the user, so the taint sink has already determined the result through explicit flow analysis.

In addition, for sensitive word detection, four sensitive words involving drugs or violence were detected through sensitive word detection. Statements such as "Your mail has arrived!" or "No one has fed the dog!" may also be user's private information, but they do not contain sensitive words. In this regard, users can customize sensitive words into the thesaurus to prevent information that they think private.

Since most IoT applications are not open source, this paper only tests the SmartThings platform and obtains less data. In addition, the effect of implicit stream analysis is not obvious, and should be combined with dynamic detection. In short, the tool will be further improved in the future and used to detect more IoT platforms.