**MANISCOPE**

0. Abstract部分：

-MANISCOPE是什么？

ManiScope是一款从官方文档自动构建Android应用清单模式并检测应用清单文件错误配置的工具，旨在解决Android应用清单文件（Manifest）中因错误配置引发的安全与兼容性问题。

--manifest是什么？&& 如何实现检测的呢？

Manifest:Android操作系统(os)中应用（app）中定义权限和组件访问规则的关键配置文件。

实现检测方式：使用自然语言处理（NLP）方法从官方的文档中提取规则并生成XML Schema(XSD)然后对其进行验证。

--- MANISCOPE效果？

应用其对Google Play 应用以及 Samsung固件中的250w个预装应用进行了检测，发现超过30%的应用都存在配置错误，其中不少都涉及不同的安全风险，eg.组件劫持、数据泄露、消息伪造;

1.Introduction部分：

-Manifest地位？

每个安卓的app都被要求在APK的根目录中有一个AndroidManifest.xml文件，这个文件要包含有关app的部分配置，eg.包名、可用Android的版本、所需要的应用组件、所需的权限或者是功能配置等内容。

有这样的要求是为了方便app的发布，方便用户简单地判断这个app能否兼容这个设备，也方便类似Google play等app商城统一管理。

也正因为它很重要，所以在一个app上架应用商城时，其Manifest需要通过例如Google Play等的多重验证。但作者发现这个多重验证并没有包含所有可能的错误，作者猜测Google play只是对app进行了过滤，以防某个设备下载到了不兼容的app。也因此，就会产生安全问题。

-Manifest是否容易出错？

并且，就连像Amazon这样顶级的供应商的app也会出现这样的问题，eg.图一展示了其某款app的存在的漏洞——purchase replay attack（购买重放攻击）。简单来说就是因为其本来是想让receiver来接受比如从Amazon的服务器传来的信息，但其错误的把权限放在了action上，导致攻击者可以伪造一个app向其发送错误的信息，让其误以为是从Amazon服务器发来的，进而可能产生虚假的订单（不付钱但依然发货）等一系列问题。

且作者对超过240w个app进行了测试，发现其超过1/3都存在大大小小的Manifest的相关问题。

-MANISCOPE具体工作流程？

ManiScope 就像是一个“文档读懂器 + 结构检查器”：

它读懂 Android 官网写的规则（靠 NLP），把规则变成标准机器可读的校验器（XSD），然后拿去验证真实的 App 配置（AndroidManifest.xml），判断有没有错。

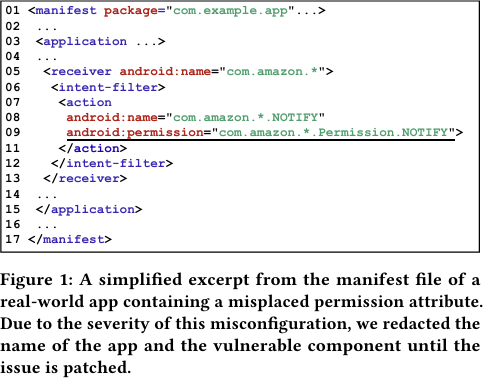
又或者说，

ManiScope 会“生成一个完整、权威的 XML 配置规则（XSD Schema）”，然后“把 app 的实际 AndroidManifest.xml 拿来对比”，看它是否符合这些规则，从而判断是否存在**配置错误**



2.Preliminaries部分

2.1 Android Manifest：XML的树状结构，其中包含层次的嵌套元素和相应的属性；下图的图1就是一个典型的XML结构

-

2.2 Manifest常见的几种错误：

-1.Misplaced element/Attribute：将属性或者是元素放在了错误位置，eg.图1将权限放在了错误的action而不是receiver；

-2.Missing：元素或属性的缺失,eg.intent-filter内缺少action

-3.Unexcepted:使用了不在文档内的无效元素，eg.拼写错误or用了foo；

-4.Wrong: 值的内容拼写错误，eg.true输入成ture

2.3 Manifest的验证方式：

既然Manifest这么重要，所以对Manifest的验证是必需的。而Google Play会对Manifest进行检查并且对特殊的元素进行过滤，以决定这个app能和什么设备兼容，缺乏系统的检查。

而现在常用的检查方式是AOSP对Manifest进行验证，但其并非对标准的XML Schema进行完整验证，而是通过手写的各种if-else方式对Manifest进行验证，是比较零散且容易疏忽的，也因此导致很多app带着错误的Manifest就上架了应用商城。

也因此，想要对Manifest有系统性的检查的话，就需要构建app系统性的XML Schema来对各个Manifest进行检查。

3.Overview部分：

对于构建XML Schema的需求，作者想从Android的官方文档提取需求并且构造一个XML Schema,总流程如下：

图示

AI 生成的内容可能不正确。

总共有五个部分：

Input: 官方文档

Module 1: 文档的收集器

Module 2: 提取文档内的嵌套关系

Module 3: 提取数量的约束，eg.一个元素底下最少有几个a元素，最多有几个b元素

Module 4: XSD的构造器，即构造XML Schema

Module 5: 根据生成的XSD对Manifest进行验证

Output: 验证结果，即Manifest是否存在问题，若存在，存在的是什么问题等。

然而这并不简单，主要挑战如下：

3.1 挑战（略写）\*

-1.识别Manifest相关的文档页面：最新版本的Android有超过1k的HTML的页面，这些不仅有关Manifest，还与eg. Android 资源的XML相关且结构十分类似。所以我们要能正确识别Manifest相关的官方页面；

-2.处理Manifest相关文件中的模糊性和不完整性：因为文档的编写是十分自由的，缺乏一个严谨的格式。因此提取结构变得有挑战性，eg.文档中类似“元素b一定要在元素a底下，且元素a最多拥有1个元素b”这样的描述，由于不是每次描述都同一个格式，所以提取面临一定的挑战。

-3.上下文领域的感知以及领域知识的解析：Maniscope在解析文档时，需要结合上下文的信息以及领域的知识共同判断真正关于manifest的结构规则，而文档常常在讲述结构性规则的时候会省略掉父元素，这就带来了挑战，需要结合上下文对规则进行补充与解析。

3.2 问题范围和假设

-范围：专注于通过使用从Android Manifest相关文档提取的约束来识别app中的manifest的意外或者是错误；

-假设：

1.从Android官方获取的Manifest相关约束都是正确且全面的；

2.输入的Manifest文件是有效的。

4.详细设计：（重点）

4.1 文档收集器：

**文档收集器**采用递归遍历算法，从Android官方文档的根元素<manifest>出发，逐层抓取子元素（如<activity>、<intent-filter>）的关联页面，过滤非清单相关内容（如API接口文档），仅保留与元素声明、属性定义及约束规则相关的章节。例如，在解析<action>元素时，自动跳过Intent机制的通用描述，专注于其特有的“canContain”“mustContain”规则。

4.2 位置约束提取器：

根据文档收集器提取到的所有文档，采用NLP技术，用位置约束提取器解析每个文档页面，分别提取 **可以包含（**“canContain”）和以及 **必须包含（**“mustContain”）部分下的属性。具体采用的是Stanford NLP句法分析器对文本进行解析并且提取出“父元素-子元素”对，用来构建manifest的结构树；例如，若文档规定<intent-filter>必须包含<action>元素，则提取器将其标记为合法子元素。为应对文档歧义性，模块引入双重校验机制：仅当实体名称存在于字典时纳入约束规则，并同步抓取元素的描述性文本（如“<action>用于声明Intent作用”）以支持后续分析。

4.3 数量约束提取器：

通过自然语言处理（NLP）技术从文档描述中提取数量约束（如最小/最大出现次数）。其核心组件包括实体识别器与约束过滤器：前者利用依存句法分析识别主谓宾关系（如从“<intent-filter>必须包含至少一个<action>”中提取父实体（主语）与子实体（宾语）），后者结合上下文（如章节标题、段落首句）和语法规则（如模态动词“必须”）过滤非清单约束。若句子包含名词性从句（acl）或状语从句（advcl），可能削弱主句的约束强度。例如，“You should always declare this attribute if you want to configure [...]”中的“if you want to configure”表明属性仅在特定条件下必需，因此主句的“must”约束无效。

手机屏幕截图

AI 生成的内容可能不正确。

更详细的流程如图5：

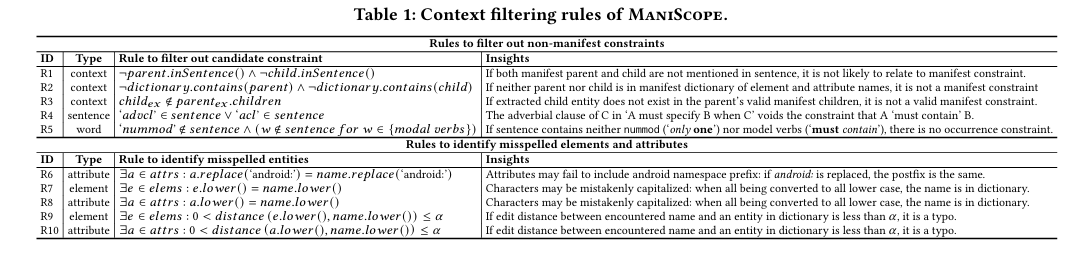
图形用户界面, 文本

AI 生成的内容可能不正确。

步骤如下：

文本

AI 生成的内容可能不正确。



具体解释：by GPT\*（如果问问题可以参考这里）

Table 1展示了ManiScope用于过滤非清单约束的规则。这些规则旨在确保提取出的约束确实是与Android应用清单文件相关的，并且能够有效减少误报。下面我将逐行解释表格中的每一项：

表格结构说明

ID：规则编号。

Type：规则类型（分为上下文过滤规则context、句子过滤规则sentence和单词过滤规则word）。

Rule to filter out candidate constraint：具体的过滤规则描述。

Insights：对规则的解释或背景说明。

具体规则解释

1. 上下文过滤规则 (Context Filter)

R1

规则：如果父元素和子元素在句子中都没有被提及，则该约束很可能与清单配置无关。

解释：确保至少有一个清单相关的实体出现在句子中，以提高约束的有效性。

R2

规则：如果父元素和子元素都不在清单元素和属性名称字典中，则该约束不是清单约束。

解释：通过检查父元素和子元素是否存在于官方文档中定义的有效清单元素列表内，进一步验证约束的有效性。

R3

规则：如果提取出的子元素不在父元素的有效子元素列表中，则该约束无效。

解释：确保子元素确实是父元素的合法子元素，防止错误关联。

2. 句子过滤规则 (Sentence Filter)

R4

规则：如果句子中含有从句（例如条件句），则会削弱主句中的约束要求。

解释：例如，“当开发者希望配置某项功能时才需要声明该属性”，这种情况下该属性并不是强制性的，因此不应被视为必需。

3. 单词过滤规则 (Word Filter)

R5

规则：如果句子既没有数量修饰词（如“仅一个”）也没有表示必须的模态动词（如“必须包含”），则不存在出现次数的约束。

解释：确保只有在句子明确指出某个元素或属性必须存在或有特定的数量限制时，才会将其视为有效的约束。

R6

规则：如果属性名缺少android:前缀但去掉前缀后匹配字典中的属性名，则认为是拼写错误。

解释：处理开发者忘记添加android:命名空间前缀的情况。

R7

规则：如果元素名转换为全小写后匹配字典中的元素名，则认为是大小写错误。

解释：处理首字母大写或其他字符大写的错误情况。

R8

规则：同R7，适用于属性名。

解释：同样处理属性名的大小写错误。

R9

规则：如果遇到的名字与字典中名字之间的编辑距离小于等于阈值α，则认为是拼写错误。

解释：通过计算编辑距离来识别可能的拼写错误，通常设置α=1以减少误报。

R10

规则：同R9，适用于属性名。

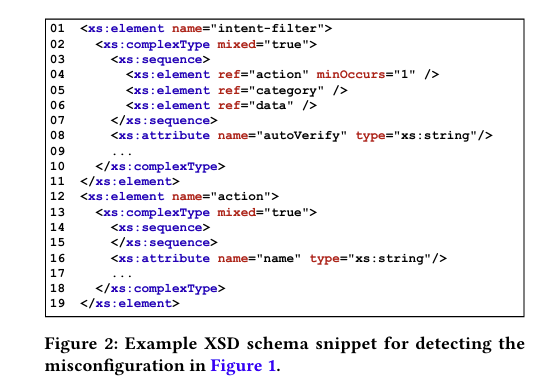
解释：同样用于检测属性名的拼写错误。

这些规则共同作用，确保ManiScope能够准确地从文档中提取出有效的清单约束，同时尽量减少误报。

4.4 模式生成器

包含2个步骤：

-位置约束的转换：将每个ele都声明为xs:element并且列出子元素以及对应属性：例如图2中的<application>元素在第1行声明，并且包含了对子元素如<activity>的引用（第4行），以及属性如android:autoVerify（第8行，实际声明在第11行）。



-定量的约束转换：设置每个element的最小出现次数 minOccurs和最大出现次数 maxOccurs，设置属性，只考虑一个——是否必需 required。

示例： （理解这个就好）

假设我们有以下位置和定量约束：

* 元素<application>可以包含多个<activity>子元素，但至少需要一个。
* 属性android:label是必需的。

这些约束会被转换成如下XSD片段：

文本

AI 生成的内容可能不正确。

理解：

首先line1:把元素application声明；

然后line4:把application的子元素activity声明 标明最少的出现次数&最多次数（最少为1 最大不限）；

然后是line6:声明必需的属性——android:label标为required。

4.5 清单验证器（Manifest Validator）：

处理三种问题：

1. **缺失实体**：
   * 如果验证器发现某个元素或属性缺失，则标记为错误。
2. **错位实体**：
   * 错位和拼写错误的实体都被报告为“意外关键词”，因此需要与清单字典进行比较。
   * 如果实体是有效的清单名称，则认为是错位；否则视为拼写错误。
3. **意外实体**：
   * 验证器会检测所有意外的属性和元素，但并非所有都是开发者拼写错误。例如，编译器可能会添加提供编译器信息的属性，或者系统独有的元素和属性不会出现在文档中。
   * 为了减少误报，只关注以下三类拼写错误：
     + **前缀错误**：例如android:package vs package，android:name vs name。
     + **大小写错误**：例如meta-data vs Meta-Data。
     + **拼写错误**：例如meta-data vs mata-data。

5.Evaluation 评估：

5.1 Manifest相关约束的提取结果：

最后提取了20份不同的Android文档对Maniscope进行测试，从最新的7.1.2到最旧的1.6版本，结果如Table 2：

表格

AI 生成的内容可能不正确。

最后获得了适用于28个element的254个manifest的约束以及125个属性。

FP && FN :

**False Positive (FP) - 假阳性**

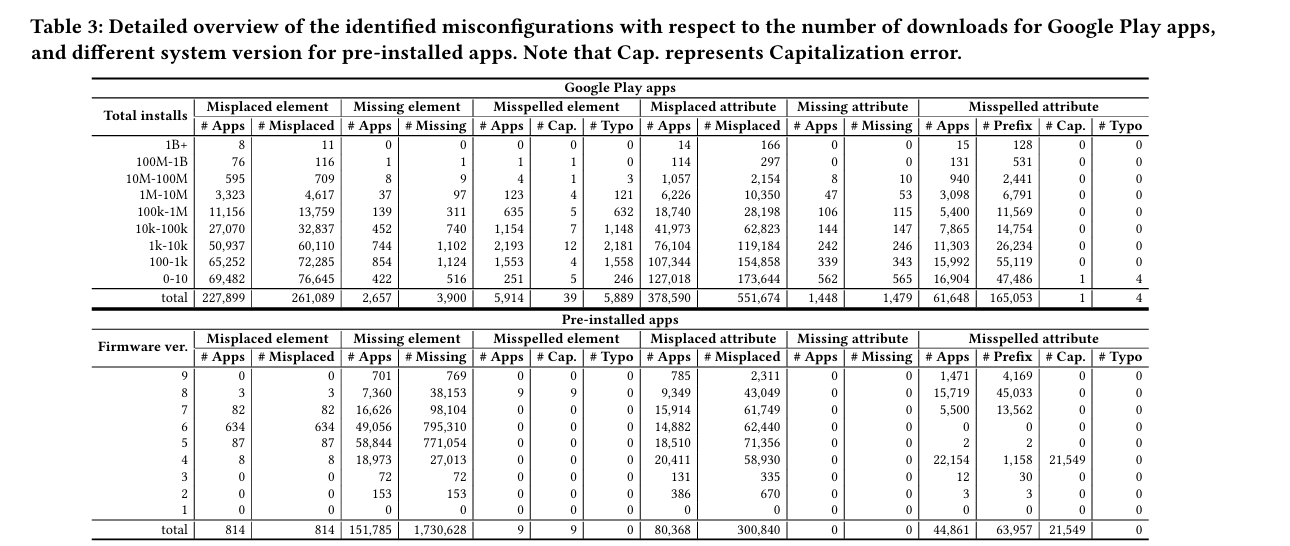
* **定义**：指系统错误地将“正常”或“正确”的条目标记为问题。简单说，就是“误报”——系统报告了一个错误，但实际上这个错误不存在。
* **在上下文中的例子**：  
  在清单验证器中，如果系统错误地将一个有效的清单元素（如正确拼写的属性）标记为“缺失”或“意外”，这就是FP。例如：
  + 规则中（如R6或R7）旨在减少FP：如果属性名缺少android:前缀但实际上是拼写错误，系统可能会误报为无效（FP），但规则通过字典匹配来避免这种情况。
  + 目标：最小化FP，以减少开发者的不必要警告和误报。

**False Negative (FN) - 假阴性**

* **定义**：指系统未能检测到“真实”的问题。简单说，就是“漏报”——系统忽略了实际存在的错误。
* **在上下文中的例子**：  
  在清单验证器中，如果系统遗漏了一个真正的错误（如一个必需属性缺失），这就是FN。例如：
  + 在4.5节中，验证器通过检查元素位置和数量约束来减少FN：如果系统未发现某个必需的属性（如android:label），就会导致FN，可能导致应用程序安全问题。
  + 目标：最小化FN，以确保所有关键错误都被捕获。

5.2 Manifest相关的错误的检测结果：

如Table 3所示：



且经过验证发现Miniscope所发现的问题内，存在2.7%的FP（根本原因是官方文档的拼写错误）但没发现任何的FN（原因是Miniscope的约束提取较严格）。

5.3 与安全相关的错误配置

分析Maniscope所发现的Manifest上的问题会带来真正的安全漏洞，一共包含四种：

**1. Exposed Component（组件暴露）**

* 原因：由于 manifest 配置错误，**没有设置权限或 export 标记错了**，导致组件（如 <receiver>、<activity>）可以被其他 App 调用。
* **后果**：攻击者可以调用内部组件触发行为，比如伪造付款广播（purchase replay attack）

**✅ 2. Permission Bypass（权限绕过）**

* 原因：**权限字段配置错位置、格式或拼写错误**
* 结果：Android 系统不会正确应用权限保护，造成无保护调用
* **后果**：攻击者可绕过权限限制调用敏感接口

**✅ 3. Intent Forgery（意图伪造）**

* 原因：组件未正确验证发送方或接受的 intent
* 漏洞点通常出现在：manifest 错配导致组件默认是 export 状态
* **后果**：恶意 App 可构造伪造 Intent，诱导目标 App 做危险操作（如跳转、打开 URL、启动服务）

**✅ 4. Data Leakage（数据泄露）**

* 原因：配置错误导致 content provider、service、broadcast 等意外暴露
* **后果**：本应受保护的数据被任意 App 读取，比如用户 ID、状态、缓存内容

Table 4展示了根据CVSS 3.1(通用漏洞评分系统)的标准，对不同Manifest配置错误所进行的安全评分的分析。

Ps:

1. 分越高代表越危险，满分十分；
2. 评价的标准是CVSS的安全维度指标：

AV = Attack Vector（攻击途径）  
AC = Access Complexity（攻击难度）  
C/I/A = 影响机密性/完整性/可用性（None/Low/High）

1. Sample Impact代表这项配置错误课内产生的后果或可能的攻击类型；

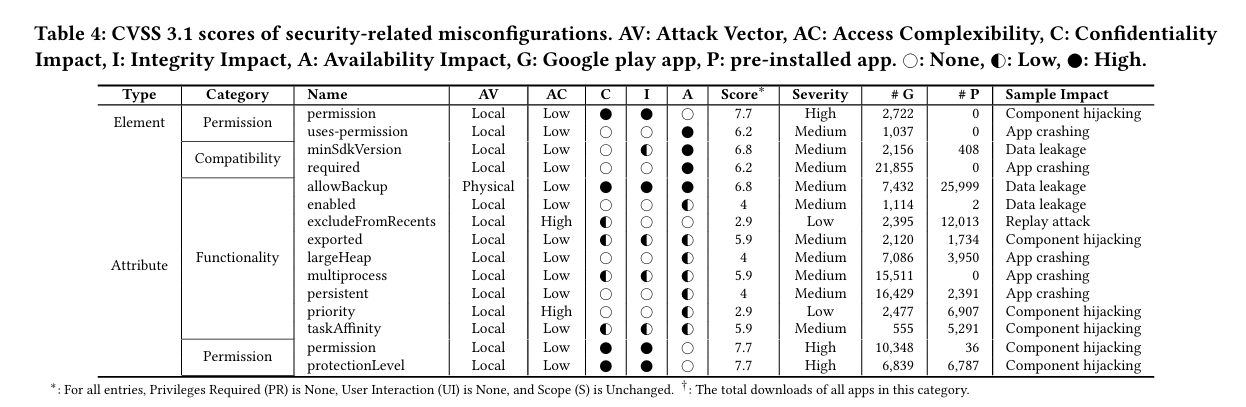


Table 5:

展示了两类来源的app——从Google Play下载的app和预先安装的app的Manifest相关错误所引起的安全问题的分布：

应用程序, 表格

AI 生成的内容可能不正确。

6. Security Case Study部分：

这部分具体解释了Manifest的配置错误会如何引起安全性的问题：

6.1 组件劫持

如果将属性 android:permission 错误地放在了action元素而不是receiver元素中的话，就会导致攻击者向receiver发送特制的内容导致权限被绕过（即前文的Amazon的问题），被恶意利用的话就会导致伪造购买行为，被用于欺诈消费。

6.2数据泄露

若将Attribute android:Backup放在错误的位置的话，就会导致数据的泄露。其应该在application元素中被设置为false，以防止app备份敏感数据，但开发中将其放在了错误位置，就可能遭受攻击并导致数据被备份并且被泄露。

6.3 频道劫持

作者发现了许多app上出现了官方文档中没有提及到的元素的应用，这些作为意外元素和属性的出现却没有被认为是拼写错误。其中一个元素名为protected-broadcast的元素本是拿来限制一些广播操作只能由系统发送，但其只适用于部分的android环境，当它在第三方应用中配置的时候，其会被忽略，也因此失去了限制作用，容易产生安全漏洞。

Table 6:

Table ABC分别对应三种问题，每个表展示了检测出来存在对应问题的前几类的app的数量，颜色显示的是下载量的总和，例如表A的第一行意思就是属于Game的app有6406个遭受组件劫持的问题，且其下载总量大于十个亿。

表格

AI 生成的内容可能不正确。

7.Discussion部分:

7.1产生这些Manifest的原因：

-开发者的疏忽

-官方文档存在错误

-错误代码的重用

7.2 可能的减缓方式

- Android主动向开发者和最终的用户显示可能的警告，以方便发现问题以及修复问题；

- Google应该提供正确且清晰的文档，确保展示的代码都是正确且安全的且为开发者提供系统的、全面的验证工具；

- 开发者应该仔细辨别代码片段，且利用自动化的工具以验证代码安全性；

7.3 Maniscope的缺陷以及未来工作

- 还没能覆盖所有的元素以及属性，只是覆盖了官方文档中的元素和属性，还没覆盖所有的元素和属性；

- 所分析的案例还不够全面，文中只是分析了3个案例。未来可以进行更全面的案例研究以识别潜在攻击并引起关注和修复问题。

7.4 肯定用不着，略

8. 相关工作

感觉也用不着 略

9. 结论 Conclusion：

提出了Maniscope——一个能从官方文档中构建Android app的Manifest清单并且将其与app的Manifest进行对比，以检测其是否存在配置错误的工具。Maniscope采用了NLP技术，并对超过240w的app进行了测试，发现超过30%的app都存在不同的问题，并且还深入分析了3个案例以揭露Manifest的错误是如何引起安全问题的。