摘要

taindroid 是一个高效的系统级的动态污点跟踪和分析系统，它能够同时跟踪多个敏感信息源。taintdroid 借助于 android 的虚拟执行环境进行实时分析。它只会占用 cpu 14%的性能，并且在同第三方 app 进行交互的时候，造成的性能损失也是微乎其微的。在使用它对 30 个著名 APP 进行监视的时候，我们发现其中 20 个 APP 里面含有 68 个可能会对用户隐私信息造成泄漏/错误利用的实例。使用 taindroid 来监视敏感数据可以将第三方 APP 的使用记录提供给手机用户，并给智能手机的安全服务企业提供宝贵的数据，以便找出 app 中的不当行为。

1 概述

在充分使用 app 的各项功能和保护用户的隐私之间，移动平台操作系统做得并不好：决定一个 app 是否可以访问隐私数据时，只提供了粗粒度的访问控制，而并没有提供一个深层次的指示：我们的隐私数据是如何被使用的。比如，如果一个用户允许某个 APP 访问它的地理位置信息，之后他是无法知道这个 APP 是否会将他的地理位置信息发送到一个基于位置的服务，或者发送给一个广告商，或者一个 APP 开发者等等。这就造成一个结果，用户必须盲目地相信 APP 将会合理使用他的隐私信息。

本论文介绍了 TainDroid 工具，它作为 android 移动平台的扩展，可以跟踪隐私敏感数据的“流动路径”。TainDroid 假设所有下载的第三方 app 都是不可信的，并实时监控这些 app 是如何访问和操作用户的隐私数据的。我们的主要目标是检查系统在什么时候通过不可信的 app 泄露了用户的隐私数据，并帮助用户或安全服务商分析该 app.

分析 app 的行为需要充分的、关于“什么数据离开了设备”和“这些数据会发送到哪”的上下文信息。所以，TainDroid 自动地给那些来自于用户敏感隐私源的信息贴上“标签”，并在这些隐私数据通过程序变量、文件或进程间通讯 IPC 进行传播的时候同时传递这些标签。当被标记的数据通过网络进行传递的时候(或者其他的，会导致这些数据离开设备的情况)，TainDroid 将会记录下数据的标签、传递该数据的 app 以及数据的目的地址。

为了能够实际部署，TainDroid 所占用的 cpu 性能必须极小。不同于现存的依靠整个系统仿真的重量级解决方案，我们借助于 android 的虚拟化结构来整合污点传播的四个粒度：变量级、方法级、消息级和文件级。我们的贡献主要在于整合了这些独立的旧技术，以及在资源有限的智能手机上对性能和精确度的合理权衡。

类似于信息流跟踪系统，TainDroid 的根本局限性在于恶意 app 可以通过隐式的信息流来泄露信息，进而绕过 TainDroid。不过这些隐式的信息流可以通过静态分析来进行检测(这将在第 8 章进行讨论)。

文章的结构如下：

①第 2 章从高层次概括了 TainDroid；

②第三章描述了 android 平台的背景信息；

③第四章描述了 TainDroid 的设计方案；

④第五章描述了通过 TainDroid 对污点源进行跟踪；

⑤第六章描述了我们对 app 的学习成果；

⑥第七章描述了我们原型实现的性能；

⑦第八章讨论了我们方案的局限性；
⑧第九章描述了相关工作；
⑨第十章总结。

2 方案简介

在智能手机上监视用户的隐私信息是否会因网络而泄露面临几个挑战：
①资源有限，相关工具为 Panorama;
②许多第三方 app 都被赋予了多种类型的隐私数据访问权限，因此，监视系统就必须区分信息的类型，而这项工作又需要额外的计算和存储空间；
③基于上下文(环境)的隐私信息是动态不定的，甚至在相近时间内发送的信息都难以鉴别区分。比如，地理位置信息是由一对浮点数表示的，这对浮点数会频繁地变化，并且难以预测。
④app 之间可以共享信息，这就会导致如果我们将监视系统限定到单个 app 的话，那么就无法兼顾到那些通过文件或 IPC 来实现 app 间隐私数据传输的情况(即被监视的 app 本身不直接发送隐私数据，而是通过文件、IPC 将隐私数据共享给另外的未受监控的 app，通过这样的方式进行隐私泄露)。

敏感信息首先会在“污点源”进行鉴定，在这里指定该敏感信息的类别。然后动态污点分析会跟踪被标记的数据是如何影响其他数据的，以找出可能会导致最初的敏感信息泄露的途径。这种跟踪通常是执行在指令层的。最后，受影响的数据将会在离开系统之前，在“污点信息终点”(taint sink，污点槽，通常是网络接口)处被鉴别。

现存的污点跟踪技术有很多限制。首先，这些依赖于指令层动态污点分析的方法会使用整个系统进行模拟，这将导致高昂的性能损耗(2-20 倍的性能损耗)。其次，开发精确的污点传播逻辑已经在 x86 指令集上被证明是相当有难度的。在实现指令层跟踪的时候，如果栈指针被错误标记的话，那么就会造成“污点爆炸”；如果复杂指令(如, CMPXCHG, REP MOV)没能正确执行的话，也会造成“污点丢失”。虽然当前绝大多数智能手机都是使用 ARM 指令集，但是跟 X86 指令集一样，也同样存在误报和漏报的问题。

表 1 描述了我们在智能机上使用污点跟踪的方法。我们利用了智能手机基于虚拟机的结构特性来实现高效的、系统范围的污点跟踪技术。此技术使用了细粒度的标记，以便更加清晰地进行语义解释。

首先我们通过 VM 解释器实现对不可信任 app 代码的变量层的跟踪。通过使用由解释器提供的语义变量，可以获取变量的上下文(环境)，这样就能避免在 X86 指令集上出现的污点爆炸现象了。另外，通过跟踪变量，我们能够实现只对数据进行标记，而不用对代码进行标记。

然后，在多个 app 之间，我们使用消息级的跟踪。我们跟踪“污点消息”，而非消息内的“污点数据”，可以极大减小在整个系统范围内进行动态分析时 IPC 的花费。

第三，对于由系统提供的 native 库，我们使用方法层的跟踪。这里，我们在执行 native 代码的时候不会插桩，只会在该方法返回的时候 patch 上污点传播信息。

最后，我们使用文件层的跟踪，以确保这些信息还保留有它们的污点标记。

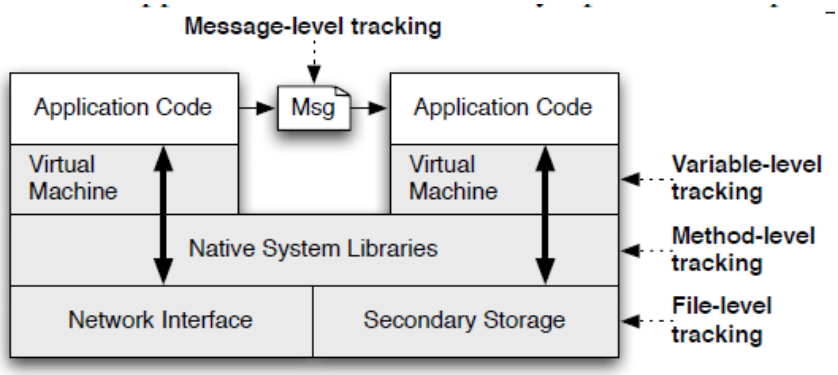

Figure 1: Multi-level approach for performance efficient taint tracking within a common smartphone architecture.

为了更方便地指定标签 taint tag，我们借助了那些定义良好的系统接口，因为 app 通常都是通过这些接口来访问敏感数据的。比如，所有从 GPS 中提取的信息都是位置敏感信息，所有从地址薄中提取的信息都是联系人敏感信息。这样，我们就可以避免通过启发式或手工分类制定的方式来制定标签了。详细的信息源介绍在第 5 章。

为了在多应用环境下实现跟踪，我们的方案要求硬件是诚实可信的。污点跟踪系统信任在用户层执行的虚拟机，以及被不可信的 app 加载的任何的由系统提供的 native 库文件。在我们的定制系统中，我们修改了 native 库加载器，以便 app 只能加载系统自带的本地库，而不能加载由 app 下载的 native 库(局限性太大了!!)。

总之，我们提出了一个新颖的，高效的，系统范围的，多种标记的污点跟踪方案，该方案主要是将现存的几种信息跟踪方法进行整合。虽然一些技术，如在解释器中实施的变量跟踪技术已经被提出(在 9 章中有讲)，但是我们的方案是第一个将它扩展到系统范围的！通过选择一个多粒度的方案，我们在性能和精确度上找到了平衡点。

3 android 系统介绍

略

4 TaintDroid 设计方案

taintDroid 在 VM 解释器内使用变量级的跟踪方法。多个污点标记 taing maskings 以一个 taint tag 的形式进行存储(一个 tag 为 32 位，每种 masking 占 1 位)。当 app 执行 native 方法的时候，将会在方法返回的时候给相应的变量(返回值)加上 taint tag。最后，这些 taint tags 会被发送到 parcels 中并通过 binder 进行传输。此论文的技术报告[17]包含有更详细的信息。

图 2 描述了 taintDroid 的结构：

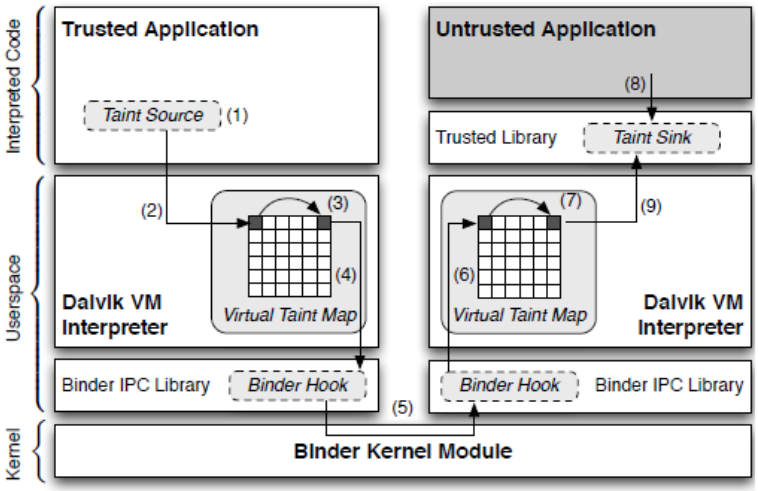

Figure 2: TaintDroid architecture within Android.

信息在一个可信的 app 中被 tainted(1)，污点接口 tain interface 调用一个 native 方法(2)，此方法 interface 了一个 DVM 解释器，在每个解释器中都有一个虚拟的污点映射表 virtual taint map，该 map 存储有特定的污点标记。当可信的 app 使用被 tainted 信息的时候，DVM 会根据我们制定的数据流规则(具体规则见后文)来传输 taint tags(3)。每一个 DVM 解释器实例都会同时传输 taint tags。当可信的 app 在进行 IPC 通信的时候，如果使用了 tainted 信息，被更改的 Binder(4)就会确保该 parcel 含有一个 taint tag，且此 tag 反映了当前 parcel 中所有数据的污点标记的综合信息。该 parcel 通过 binder kernel module(5)进行传输，然后被远程的不可信 app 接收。注意，只有翻译码是不可信的！更改后的 binder 会从 parcel 中抽取出 taint tag，并进行相应的赋值操作(6)。远程的 DVM 解释器实例同样会将 taint tags 传输给不可信 app(7)。当一个不可信的 app 调用一个动态库，而该动态库又被指定为一个污点槽的时候(如，网络传输)，该库就会从待发送数据中抽取出 taint tag，并报告此事件。

要完成此结构，需要克服几个挑战，包括：

①taint tag 存储

②翻译码的污点传播

③native 码的污点传播

④IPC 的污点传播

⑤二次存储的污点传播

4.1 taint tag 存储

这涉及到性能和内存消耗。动态污点跟踪系统通常为每一个数据字节或字存储一个 tag。被跟踪的内存是非结构化的，并且没有语义内容。通常，taint tags 存储在非邻近的影子内存和 tag maps 中。taintDroid 在 DVM 解释器中使用可变的语义。我们在内存中将 taint tags 存储在变量相邻的位置，以实现空间局部性。

dalvik 有 5 种类型的变量需要进行污点存储：方法的本地变量，方法的参数，类的静态域，类的实例域，数组。在所有情况中，我们给每一个变量存储一个 32 位的比特向量 bit vector，该 bit vector 用来编码 taint tag，这样每个 tag 就允许有 32 种不同的 taint masking 了。

Dalvik 在内部栈中存储方法的本地变量和参数。当 app 调用一个方法的时候，虚拟

机会为该方法分配一个新的栈帧。方法的参数也是通过内部栈进行传递的。在调用一个方法前，被调用者会使用传递进来的参数替换栈的顶部数据，这样这些参数就可以在被调用者的栈帧中拥有高标号的寄存器了。我们通过倍增栈帧的方法来分配 taint tag 的存储空间。taint tag 交叉存储在各个变量之间。如未修改之前，使用 fp[i]来访问寄存器 v[i]，修改之后，就需要使用 fp[2i]来访问了。注意 Dalvik 使用两个相邻的 32bit 寄存器来存储一个 64bit 的变量，所以在修改之后，我们就使用 fp[2i]和 fp[2i+2]来进行 64bit 数据的存储和读取操作。最后，native 方法所需要的栈帧也会进行类似的修改，只不过会有一点点不同，详情见 4.3。图 3 展示了更改后的栈帧格式：

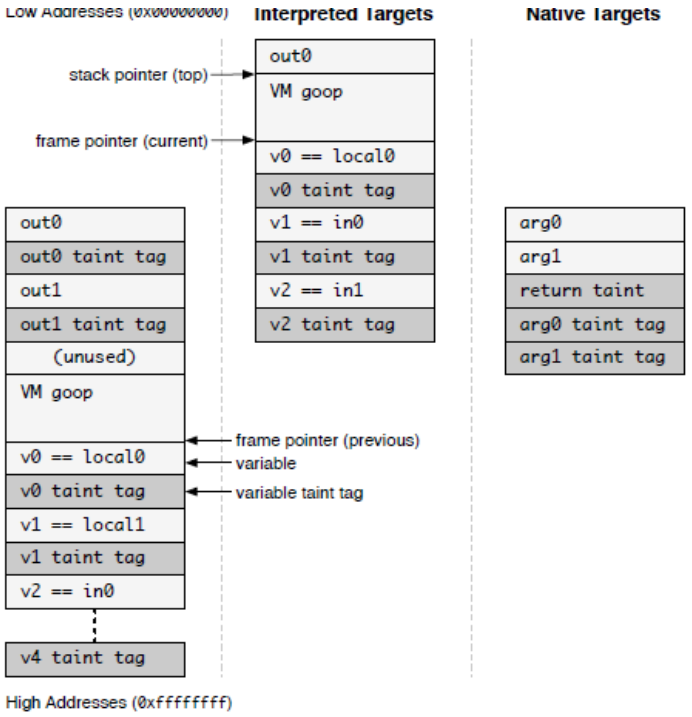

Figure 3: Modified Stack Format. Taint tags are interleaved between registers for interpreted method targets and appended for native methods. Dark grayed boxes represent taint tags.

需要注意的是，taintDroid 对每一个数组只存储一个 taint tag，以减小内存消耗。当然这样做，肯定会在污点传播的时候发生误报。比如，如果一个未被 tainted 的变量 u 存储在数组 A 的 index 0 处(A[0])，而一个被 tainted 的变量 v 存储在 A[1]，那么 A 就是被 tainted 了。之后，如果变量 v 被移到了 A[0]，但是由于 tag 并不会移动，所以此时 u 的 tag 就变成了 v 的 tag，也就是说 v 也会被错误地 tainted。幸运的是，java 频繁地使用对象，并且对象引用并不经常被 tainted(4.2 有讲)，所以这种编码方式在实际测试中带来的误报率很低。

4.2 翻译码污点传播

污点跟踪的粒度和流机制会影响性能和精确度。变量为污点传播提供语义变量 valuable semantics，用于从标量中区分数据指针。taintDroid 主要跟踪基本类型的变量(int, float 等)，但是，在必要情况下，为了确保污点传播正确执行，也会对对象引用进行 tainted。本节将会重点介绍为什么这些“必要情况”是存在的。

4.2.1 污点传播逻辑

鉴于 DVM 独特的结构，所以我们需要为其创建对应的传播逻辑。我们使用之所以要使用数据流逻辑，是因为在跟踪隐含流的时候需要进行静态分析并且还会造成重大的性能损耗等(在第 8 章有讲)。我们首先定义 taint masking，taint tag，变量和污点传播。然后，描述我们对 dex 文件创建的污点传播逻辑规则。

ps:由于没安装 MathType，所以无法正确打印表 1 中的符号，大家这里可以参照原文。且，为了更好地理解本节，一定要熟悉 smali 指令。

设定符号Σ 表示一个特定系统中的所有 taint masking 的总和。一个 taint tag *t* 作为 taint masking 的集合。显然 t ∈Σ 。每一个变量都含有一个对应的 t。一个变量是前面在 4.1 节中描述的 5 种变量类型之中一种。我们对每种类型都使用不同的表述方式：本地变量和参数属于“虚拟寄存器”，用 v_x 来表示；类的域变量用 f_x 来表示，其中的 x 表示具体的类；实例域需要一个实例对象，所以用 $v_y(f_x)$ 来表示，这里 v_y 代表实例对象的引用(注意，对象引用和被引用的值都是变量)；静态域用 f_x 表示，这是 $S(f_x)$的简写，其中 S()表示一个静态范围；最后 $v_x[.]$表示一个数组，其中 v_x 是一个数组对象的引用。

我们定义虚拟污点映射 taint map 函数是π (.)。π (v)返回变量 v 的 taint tag t，它也可以用作为变量 v 分配一个 t，可以通过符号←来进行区分。当π (v)在←右边的时候，π (v)表示获取 v 的 t；反之，表示为 v 分配一个 t。比如π (v_1) ←π (v_2)，表示将 v2 的 t 赋值给 v1 的 t。

表 1 描述了我们的污点传播逻辑。其中 R 和 E 分别表示在解释器中保存的返回和异常变量。A,B,C 是字节码中的常量。该表并没有列出那些会清除目的寄存器中 taint tag 的指令，比如，我们不认为 array-length 指令会返回一个被 tainted 的值，即使该数组是 tainted 的(这个很好理解，比如，敏感数据的长度本身并不是敏感的)。不过注意，数组的 length 有时候有助于直接控制数据流的传播。

Table 1: DEX Taint Propagation Logic. Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables maintained within the interpreter. A, B, and C are byte-code constants.

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

4.2.2 tainting 对象引用

表 1 所示的传播规则明显有 2 个“不同寻常的地方”(aget-op 和 iget-op，因为它们都额外考虑了索引 Vc 的 tag)。

第一，在处理转换表的时候(如，ASCII,UNICODE 之类的转换)，污点传播逻辑通常会考虑到数组索引的 taint tag(即 aget-op 中的 Vc 的 tag)。比如，考虑这么一个转换表：将小写字母变大写。如果一个 tainted 变量值"a"是一个数组的索引(即

Vc)，那么在转换之后，即使本来数组中的"A"不是一个 tainted 的变量，但是我们获取的返回值"A"(即 Va)也应当被 tainted。因此，为了实现前述的思想，aget-op 指令的污点传播逻辑同时使用数组与数组索引的 tag。再举一个具体地例子，有隐私信息“chouchou”，假设不考虑索引的 tag，那么如果恶意 app 将它转换为“CHOUCHOU”，就意味着丢失了 tag，这不就意味着隐私信息变为了非隐私信息了么！而如果考虑了索引的 tag，就大不一样了。

第二，当数组中包含有对象引用的时候(如，一个 interger 数组)，那么索引的 taint tag 就会被传播到对象引用，而不是对象值。所以，在 get(iget-op)规则中我们定义：当由对象引用读取对象值的时候，要将对象引用的 taint tag 传递给对象值。

在图 4 中的代码展示了一个真实的场景，它表明对对象引用进行 tainting 是必要的！

```
public static Integer valueOf(int i) {
  if (i < -128 || i > 127) {
    return new Integer(i); }
  return valueOfCache.CACHE [i+128];
}
static class valueOfCache {
  static final Integer[] CACHE = new Integer[256];
  static {
    for(int i=-128; i<=127; i++) {
      CACHE[i+128] = new Integer(i); } }
}
```

Figure 4: Excerpt from Android's Integer class illustrating the need for object reference taint propagation.

这里 valueof 函数接收一个 Int i，返回一个 Interger 对象。如果该 int 值在-128 到 127 之间，那么 valueof 函数就会返回一个静态定义 Interger 对象的引用。考虑如下定义的方法 intProxy 及其使用：

```
Object intProxy(int val) { return val; }
int out = (Integer) intProxy(tVal);
```

假设 tVal 是一个 int 变量 1，并具有 taint tag TAG。当 intProxy 传递 tval 的时候，TAG 就会传播给 val。当 intProxy 返回 val 的时候，它会调用 integer.valueof() 函数，以获取标量 val 对应的 integer 实例。在本例中，integer.valueof 返回一个静态 integer 对象的引用（valueofCache.CACHE[129]），且对象值为 1。此时该对象值的 taint tag 为空，但是，根据 aget-op 的传播规则，该对象引用会包含索引寄存器的 tag，即该对象引用就会含有一个污点 TAG。所以只有在从 Integer 中读取对象值的时候(即通过 iget-op 从对象引用中获取对象值)就将对象引用的 taint tag 传递给对象值，这样才能将正确的 TAG 传递给 out 变量。

4.3 native 代码污点传播

native 代码在 TaintDroid 中是不受监控的。理想情况下...因此，为了在类 JAVA 环境中更精确地进行跟踪，我们定义了两个必要的后置条件：1)所有被访问的 external 变量都要按照数据流的规则分配 taint tag(如类的域被其他方法引用)；2)返回值也同样按照数据流的规则分配一个 taint tag。

TaintDroid achieves these postconditions through an assortment of manual instrumentation, heuristics, and method profiles, depending on situational

requirements.(不知道如何正确翻译~)

内部的 VM 方法：这些方法由被解释后的代码直接调用(传递一个指向 32 位寄存器参数数组的指针，返回一个指向返回值得指针)。鉴于在这些方法中只有较少的方法才会进行频繁的版本更新，我们会在需要的时候对其进行手工修补。

JNI 方法：这些方法由 JNI 桥进行调用。为了实现污点传播，我们对 JNI 调用桥进行了修改。当一个 JNI 方法返回的时候，TaintDroid 会询问一个方法 profile 表，该表用于传播信息的更新。一个方法 profile 是一个(from,to)数据对的 list，用于指示变量之间的数据流(这些变量可能是参数、类的变量、返回值等)。枚举所有 JNI 方法的信息流是很耗时的，最好通过源代码分析进行自动化实现(这是我们下一步工作)。我们当前包含了一个额外的传播启发式补丁。就 JNI 方法而言，这个启发是很保守的——只有在原始的、Sting 参数和返回值中使用。它将方法参数的 taint tags 联合在一起分配给返回值得 taint tag。

4.4 IPC 污点传播

当 app 之间进行数据交换的时候，其 taint tag 也必须进行传播。这就是消息级的污点跟踪。一个消息的 tait tag 代表了该消息包含的所有变量的 taint masking 的上界。

为什么要进行消息级的污点跟踪呢？试想这么一个场景：如果一个 IPC parcel 消息包含有一系列的标量，那么接收者就可以通过解包消息 string 来绕过变量级的污点传播机制，因为这样的话，接收者就可以获取自己想要的数据，且不用传播该消息中所有的变量的 taint tag。

但是消息级污点传播会造成误报。类似于数组，在 parcel 中所有的数据项都共享一个 taint tag。比如，在 8 章中讨论的跟踪 IMSI 的局限性。

4.5 二次存储的污点传播

taint tag 可能会在某个数据被写进文件的时候丢失。因此我们为每个文件建立一个 taint tag。当文件进行写操作的时候，我们会对该文件的 taint tag 进行更新，当读取文件的时候，我们也会将该文件的 taint tag 传播给读出来的数据。TaintDroid 将文件的 taint tag 存储在文件系统的扩展属性中。为了实现这一点，我们为 android 的主机文件系统(YAFF32)实现了扩展属性的支持，并将 SD 卡格式化为 ext2 文件系统。正如前面说过的数组和 IPC 一样，为每个文件设置一个 taint tag 会导致误报现象，并且会限制信息数据库的 taint masking 的粒度。当然，我们可以在增加内存和性能消耗的情况下提升污点跟踪的粒度。

4.6 污点接口库(interface library)

在虚拟化环境中定义的污点源和污点槽必须通过我们的跟踪系统来传播 taint tags。我们将污点源和污点槽抽象化为一个单一的污点接口库。该库含有两个功能：1)为变量增加 taint masking；2)从变量中抽取 taint masking。该库只提供添加 tag 的能力，而不能设置或清除 tag(因为这些功能可能会被恶意利用)。

显然，对于 array 和 string 而言，可以通过内部的 VM 方法来为他们添加 taint tags，因为两者都是存储在数据对象中的。另外，基本类型的变量是存储在 VM 的栈中的，并且会在方法返回后被销毁。所以污点库使用方法的返回值作为基本类型的变量的污点。开发者将一个值或者变量传递进对应的用于添加污点的方法中(如 addTaintInt)，这个方法会返回同样的值或变量，只不过这个值或变量已经

被附加上了指定的 taing tag。注意，堆栈存储并不会因为进行 taint tag 检索而产生不良的并发症。

5 Privacy Hook Placement

使用 TaintDroid 进行隐私分析，需要在操作系统中对隐私敏感信息源和设备污染源进行鉴别。之前，动态污点分析系统会假设污点源和污点槽布局是无关紧要的。但是，如 android 之类的复杂的操作系统会通过多种方式给应用程序提供信息，比如直接访问，服务接口等。我们必须仔细认真地了解每一种可能的隐私敏感信息，这样才能找出一个最好的定义污点源的方法。

污点源只能在 TaintDroid 存储 tag 的内存中添加 tag。当前，污点源和污点槽主要分布在解释码的变量、IPC 消息和文件中。本章将会讨论这些有价值的污点源和污点槽在这些限制(上述 3 种分布)下，是如何实现的。我们基于信息的特征概括出如下几种污点源。

1，低带宽的传感器：大量的隐私信息都是通过这些传感器获取的，如位置传感器和加速器。这些信息经常变换，且同时被多个 app 使用。所以，智能手机操作系统一般都会使用一个传感管理器来实现对低带宽传感器的多路访问。这个传感管理器很显然是一个理想的 hook 对象。在我们的分析中，我们 hook 了 android 系统的 LocationManager 和 SensorManager 应用程序。

2，高带宽的传感器：麦克风，照相机等都是高带宽的传感器。这些传感器通常会传输大量的数据，且这些数据一般是由某一个 app 使用。所以，智能手机操作系统通常会通过一个大的数据缓存器、文件或两者兼用，来共享这些传感器返回的信息。当通过文件来共享这些传感器信息的时候，这些文件必须被标记上合适的 tag。归功于灵活的 APIs 机制，我们同时 hook 了麦克风和照相机所使用的大的数据缓存器及文件，以便实现污点跟踪。

3，信息数据库：诸如电话薄和短信之类的可共享信息通常都存储在基于文件的数据库中。这种组织结构使得我们可以将这些数据库也看做类似于传感器的污点源。通过给每个数据库文件添加一个 tag，可以实现在从这些数据库文件中读取信息的时候，读取出来的信息自动的被添加上污点 tag。需要注意的是，虽然 TaintDroid 的文件级粒度的污点跟踪适合于这类有价值的信息源，但是，对于那些很大的文件，就不太适用了。幸好，我们并没有遇到过这种情况。

4，设备鉴别：可以标志手机或用户身份的独一无二的信息也是隐私信息。并非所有的个人身份信息都可以轻易地被加上污点。但是，手机包含有几个很易加上污点的信息：电话号码，IMSI/ICC-ID，IMEI。这些信息都可以通过定义好的 APIs 进行访问，所以我们对这些 API 进行了插桩。我们将在第 8 章讨论 IMSI 污点源本身固有的局限性。

5，网络污点槽：当污点信息通过网络接口传递出去的时候，我们就会进行隐私信息的分析和鉴别。基于解释器的 VM 方法需要将污点槽安置在解释码中。因此，在 Java 框架库调用 native socket 库的时候，我们进行插桩操作。

6 应用程序研究

略

8 讨论

8.1 方案设计上的局限性

TaintDroid 只跟踪数据流(即明确的流)而不跟踪控制流(不明确的)，这样最大限度减少性能损耗。虽然 TaintDroid 可以进行隐私监控并报告，但是对于真正的恶意软件而言，它可以研究我们的系统，然后通过控制流来泄露隐私敏感信息。要想实现控制流跟踪，就需要进行静态分析，但在无法获取源码的情况下，是很难静态分析的。如果一个污点范围是可以确定的，那么就可以动态跟踪直接控制流；但是，DEX 文件并没有明确的分支结构，难以为 TaintDroid 所用。请求式的静态分析，可以得到方法的控制流程图(control flow graphs, CFGs)，该流程图提供了上下文；但是，TaintDroid 目前并没有采用这些分析方法，是为了避免误报及，及处于性能上的考虑。我们的数据流污点传播逻辑是由现存的著名的机种污点跟踪系统组成的。最后，一旦信息离开了手机，服务器一般会返回一些应答数据，而 TaintDroid 并不能跟踪这些数据。

8.2 实现上的局限性

主要是由于 DirectBuffer 对象使用的 PlatformAddress 类中的 native address。文件和网络相关的 IO 操作中，那些会对直接变量(direct variants)进行读写操作的 APIs 会假设这些 native address 是来自于 DirectBuffer 的。而鉴于 DirectBuffer 数据是以非透明的本地数据结构的方式进行存放的，所以 TaintDroid 并没有对这些 DirectBuffer 对象进行污点跟踪。不过当对这些对象进行读写操作的时候，TaintDroid 会进行提示。

8.3 污点源的局限性

虽然 TaintDroid 效率不错，但是当跟踪的消息含有配置标识符(configuration identifiers)的时候，它也会产生重大的误报现象。比如 IMSI 由 MCC, MNC, MSIN 组成，且这几个都是被 tainted 的。当 android 进行数据交换的时候，通常会使用 MCC 和 MNC 作为配置参数，而这会导致在 parcel 中的所有信息都会被 tainted——也就是说会产生污点信息爆炸。所以，理论上来说，对于那些含有配置参数的污点源，在 parcel 中单独地对各个污点变量加上污点更为合适。但是，如 6 章所示，消息级的污点跟踪机制对于大多数的污点源，工作起来效率更高。

9 相关工作

略