Android论文阅读

1.Malton

这个其实一个Android Malware的分析框架，其实整个框架搭建的非常全面和完善。确实也体现出四大卷的特性了。

Contribution and Motivation

这个论文主要的贡献点在于

1. multi-layer monitoring and information flow tracking
2. efficient path exploration

这两个贡献点的必要性作者在introduction中也介绍过了，通过阅读motivating example也可以看出了。问题出在哪里呢？（如图，这应该是某个malware的片段或者作者自己编的某个片段。）

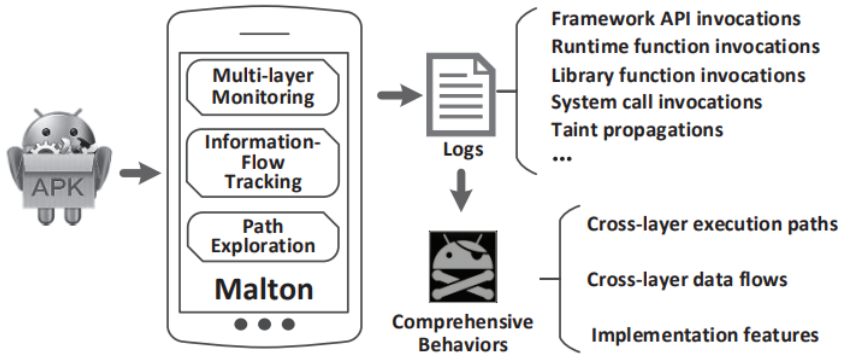
- 可以看到SMS在收到特定电话号码（controller ”6223**60“）发来的信息的时候，就会调用procCMD。根据攻击者的命令cmd来进行不同的操作。其实估计很多动态下发的dex或者云控的行为都是通过分支处理来进行的，和这个多个if-else很类似。这里存在触发行为的难度，如何能够触发cmd为1-5所有的行为，是一个问题。因此作者说要通过一个offload的动态符号执行（安卓只要收数据，服务器进路径约束求解）来进行path exploration。同时，还提供由用户直接执行某个路径的方案。
- 此外，不同cmd导致的行为是在不同层发生的，比如readIMSI是在framework层中获取的信息，通过sys_sendto在系统层发出SMTP包。而parseMSG又是个JNI方法会进入到ART层。所以作者要提出这个multi-layer monitoring and information flow tracking【跨层的监控和污点分析】，然后说现有的很多工作都是单层的，比如CopperDroid之类的。

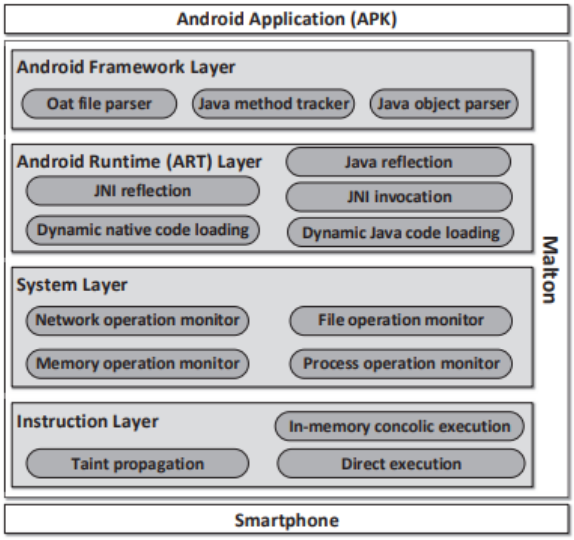
Listing 1: A motivating example

```
public static native void readContact();
public static native void parseMSG(String msg);
private void readIMSI(){
  TelephonyManager telephonyManager =
    (TelephonyManager) getSystemService(
      Context.TELEPHONY_SERVICE);
  String imsi = telephonyManager.getSubscriberId();
  // Send back data through SMTP protocol
  smtpReply(imis);
}
private void procCMD(int cmd, String msg){
  if(cmd == 1) {
    readSMS();  // Read SMS content
  } else if(cmd == 2) {
    readContact();  // Read Contact content
  } else if(cmd == 3) {
    readIMSI();  // Read device IMSI information
  } else if(cmd == 4) {
    rebootDevice();  // Reboot the device
  } else if(cmd == 5) {
    parseMSG(msg);  // Parse msg in native code
  } else { // The command is unrconginized.
    reply("Unknown command!");
  }
}
public boolean equals(String s1, String s2) {
  if(s1.count != s2.count)
    return false;
  if(s1.hashCode() != s2.hashCode())
    return false;
  for(int i = 0; i < count; ++i)
    if (s1.charAt(i) != s2.charAt(i))
      return false;
  return true;
}
public void onReceiver(Context context, Intent intent){
  String body = smsMessage.getMessageBody();
  // Get the telephone of the sender
  String sender = smsMessage.getOriginatingAddress();
  // Check if the SMS is sent form the controller
  if(equals(sender, "6223**60")) {
    procCMD(Interger.parseInt(body), body);
  }
  ...
}
```

Framework

这个架构给人一种感觉，像是先收集log，事后再进行全栈的关联？可以看到在一个APK运行时可能涉及到的接口位置，作者都做了记录。确实是multi-layer





关于记录的数据原文的说法是：Malton generates **logs** containing the information of method invocations and taint propagations at different layers and the result of concolic executions. [接下来这部看起来像是人工] Based on the logs, we can recon□struct the execution paths and the information flows for characterizing malware behaviors.

接下来，作者介绍了，每一层都关注了哪些地方，收集了哪些数据，以及如何进行关联。

Android Framework Layer

基于Valgrind对native instructions (ARM) 插桩, 需要理解arm指令序列对应的框架层的API调用

- bridge the semantic gaps between the low level native instructions and upper layer Java methods.
- 做法感觉是基于DBI的插桩，把源于Dalvik code的ARM指令通过Valgrind导出为VEX IR，然后基于Valgrind提供的一些API接口，进行IR Block的块级插桩。
- 此外，为了获得Java函数的入口，作者实现了OAT File Parser来解析OAT文件，主要的解析原理就是下图。从而可以获得methods的name和对应ARM指令的offset之类的信息（记录在hashtable，说白了就是有了函数名和函数首地址的关系）

The OAT file has an oatdata section, which contains the information of each class that has been compiled into native code. The native code resides in a special section with the offset indicated by the oatexec symbol. Hence, we can find the information of a Java class in the oatdata section and its compiled native code through the oatexec symbol.

When an app is launched, the ART runtime parses the OAT file and loads the file into memory. For each Java class object, the ART runtime has a corresponding instance of the C++ class Object to represent it. The first member of this instance points to an instance of the C++ class Class, which contains the detailed information of the Java class, including the fields, methods, etc. Each Java method is represented by an instance of the C++ class ArtMethod, which contains the method's address, access permissions, the class to which this method belongs, etc. The C++ class ArtField is used to represent a class field, including the class to which this field belongs, the index of this field in its class, access rights, etc. We can leverage the C++ Object, Class, ArtMethod and ArtField to find the detailed information of the Java class, methods and fields of the Java class.

- 之后在native instructions -> IR Block的过程中，判断每个IR Block的起始地址是否hit hashtable，如果是就插入需要记录的逻辑————记录函数调用及其参数。
- 而对于函数结束地址，ARM架构都记录在lr寄存器中，在进入起始地址的桩点逻辑中预先记录，之后在每个block退出点比对jmp到的next block地址与记录值是否一致，一致则插入逻辑即可（记录返回值）。【哈哈，和我之前实现二进制内存检查的时候，hook堆块的分配释放函数的方法几乎是一模一样】
- 当然由于参数涉及到no-primitive data，因此作者还实现了JAVA Object Parser来解析一些参数中的Java Object类型，如String Object
    - 这里有一个疑问，如果是自定义类型，不知道实现统一的解析是否可行。
- 讲真，看到这里，虽然是非常工程的实现，可能不带有多少学术意义，但是描述的真的非常详细，基本上可以完成复现。唯一担心的就是DBI本身的有效性和可行性。

Android Runtime Layer

为了监视一些更隐秘的行为（that cannot be monitored by the Java method tracker in the Android framework layer），**Malton further instruments the ART runtime (i.e., libart.so).**

- For example, the packed malware may use the internal functions of the ART runtime to load and execute the decrypted bytecode directly from the memory. Malicious payloads could also be implemented in native code, and then invoke the privacy-concerned Java methods from native code through the JNI reflection mechanism. or JNI reflection.
- 作者着重考虑了对以下函数进行插桩：

Table 1: Runtime behaviors related functions.

Behavior	Functions
Native code loading	*JavaVMExt::LoadNativeLibrary()*
Java code loading	*DexFile::DexFile()* *DexFile::OpenMemory()* *ClassLinker::DefineClass()*
JNI invocation	*artFindNativeMethod()* *ArtMehod::invoke()*
JNI reflection	*InvokeWithVarArgs()* *InvokeWithJValues()* *InvokeVirtualOrInterfaceWithJValues()* *InvokeVirtualOrInterfaceWithVarArgs()*
Java reflection	*InvokeMethod()*

- 原因：For instance, malicious payloads implemented in native code could invoke framework APIs using JNI reflection. Java reflection is commonly used by malware to modify the runtime behavior for evading the static analysis [61]. For example, framework APIs could be invoked by decrypting the method names and class names at runtime using Java reflection.

System Layer

Malton tracks system calls and system library functions at the system layer. To track system calls, Malton registers callback handlers before and after the system call invocation through Valgrind APIs.

- Valgrind提供了hook function/syscall的API，所以直接使用进行function wrapping就好。作者在选择Hook的API类型主要考虑了：Network、File、Memory、Process几个维度。

  - Network operations. Since malware usually receives the control commands and sends private data through network, `Malton` inspects these behaviors by wrapping network related system calls, such as, *sys_connect()*, *sys_sendto()*, *recvfrom()*, etc.
  - File operations. As malware often accesses sensitive information in files and/or dynamically loads malicious payloads from the file system, `Malton` records file operations to identify such behaviors.
  - Memory operations. Since packed malware usually dynamically modifies its own codes through memory operations, like *sys_mmap()*, *sys_protect()*, etc., `Malton` monitors such memory operations.
  - Process operations. As malware often needs to fork new process, or exits when the emulator or the debug environment is detected, `Malton` captures such behaviors by monitoring system calls relevant to the process operations, including *sys_execve()*, *sys_exit()*, etc.

- 最重要的file没说，因为file的操作太多了而且调用很频繁，不知道作者会怎么选择和filter。

此外，为了code exploration考虑（可能是因为基本来自于网络侧，或调试环境测试ptrace之类的），作者还会修改一些syscall的参数和返回值。但是对于路径探索的事情，作者只在syscall/system library function中进行了讨论，在Java层完全没提到，所以纯Java侧的路径探索应该是没有实现的。

- For example, the C&C server may have been shut down when malware samples are being analyzed. In this case, Malton replaces the results of the system call sys connect() to success, or replaces the address of C&C with a bogus one controlled by the analyst to trigger malicious payloads.

Instruction Layer: Taint Propagation & Path Exploration

在这个部分作者终于讲到了如何实现多层/全面的Taint Propagation。起初一直以为要进行污点信息同步之类的。

实际就是在指令层上实现所带来的好处-> To propagate taint tags **across different layers**, Malton works at the instruction layer because the codes of all upper layers become ARM instructions during execution.

直接在ARM指令上做污点分析（byte-precision），一定是包含了所有上层的逻辑的。而Souce点和Sink点的选取，作者本质是说选择了JAVA和C函数的方法中的参数和返回值，其实就是之前的“多层”桩点中（本质就是IR）加一些设置。

讲白了这篇文章就是使用了Valgrind在VEX IR上提供的一些功能完成了一个工程项目，污点分析、插桩之类的。回过头来思考，之前说的那么多层的实现，可能就是套了个"帽子"。本质都是在DBI上进行的，全都是基于指令层的插桩记录操作，只不过确实记录了上层视角中包含的一些信息。

在污点分析上，剩余有价值的东西，可能是具体选择了哪些点吧：

s of sink methods. By default, at the framework layer, 11 types of information are specified as taint sources, including device information (i.e., IMSI, IMEI, SN and phone number), location information (i.e., GPS location, network location and last seen location) and personal information (i.e., SMS, MMS, contacts and call logs). `Malton` also checks the taint tags of the arguments and results when each framework method is invoked. In the system layer, `Malton` takes system calls *sys_write()* and

sys_sendto() as taint sinks by default, because the sensitive information is usually stored to files or leaked out of the device through these system calls. As malware can receive commands from network, `Malton` takes system call *sys_recvfrom()* as the taint source by default. Note that `Malton` can be easily extended to support other methods as taint sources and sinks in both the framework layer and the system layer.

之后，作者进行了路径探索，从篇幅上看，这应该是一个重点，也是我在阅读前最想看到的部分。

To trigger as many malicious behaviors as possible for analysis, Malton employs the efficient path exploration technique, which consists of taint analysis, in-memory concolic execution with an offloading mechanism, and direct execution engine.

- taint analysis：之前的污点分析已经可以知道特定source输入会产生的执行路径
- in-memory concolic execution with an offloading mechanism：这里作者为移动端做了优化，把符号执行中比较难求解的/需要算力的部分移动到服务器或PC上，然后进行concolic execution
- direct execution engine：对于前者求解失败（无法产生感兴趣路径的input的时候），作者采用强制执行的方式

作者说基本的部分不说了，就讲讲优化和自己实现不同的地方。

1. Concolic Execution: Offloading Mechanism
    - 作者说就优化后的KLEE而言，约束求解的实际都要占到41%，所以搬到桌面系统上使用算力是合理的
    - 不过方法很矬，直接让Malton输出求解符号到logcat，让PC通过ADB读取求解约束（用的是Z3求解器）之后，把求解结果通过文件传给手机，作为输入。由于可能求解的结果很多，每个input都单独作为一个file push上去，直到push上去一个empty file，作为此次求解的标志【确实很挫】
2. Concolic Execution: In-memory Optimization
    - 由于用户可以选择一些code region【可能是很多block的区域】，Malton将会在这个区域中进行路径探索【估计为了避免路径爆炸，同时也能专注特定的部分进行快速探索】
    - 即使在选定了code region之后，路径的可能也很多，作者不希望从头开始执行整个程序，而是从感兴趣的code region处开始重新执行。但是，这就需要进行现场恢复。而作者做了这样的几件事情：
        - 确定interesting code region的入口位置【input来自哪里】以及出口位置
            - select the IR statement that lets the input have concrete values as the entry point of the code region, and choose the exit statement (i.e., Ist Exit) or the next statement (i.e., Ist Next) of the subroutine as the exit point of the code region
        - 先让样本进行执行，首先会记录入口点的寄存器状态【VexGuestArchState】，并在执行过程中记录内存store导致的修改，同时替换系统的alloc/free实现，使得之后能够free the allocated memory or re-allocate the freed memory。也就是在内存，寄存器上能实现reverse。

        - 然后到样本执行到退出点的时候，记录符号约束+offload约束求解得到新的输入，并通过预先的插桩修改程序计数器，让执行回到入口点。然后reverse现场，探索新的input。

3. Direct Execution
    - 对于那些约束求解很困难的情况，比如float-point operations and encryption routines，约束求解可能会失败。
    - 因此，**For the conditional branches with unresolved constraints**, Malton has the capability to directly execute certain code paths.
    - The direct execution engine of Malton使用了两种技术【旧货新包装】
        - a) modifying the arguments and the results of methods, including library functions, system calls and Java methods;
        - b) setting the guard value（0 or 1） of the conditional exit statement (i.e., Ist Exit). The guard value is the expression used in the Ist Exit statement to determine whether the branch should be taken
            - 在Valgrind IR中，Ist Exit被定义为`if(t) goto <dst>`，其中的t可以控制分支跳转，也就是guard value。
    - 这个部分其实写的很含糊，比如对于程序而言如何知道要往哪个分支步进？可能需要做反向搜索，从用户指定的终点IR block反向搜索到入口点，从而确定分支路径。或者可能需要之前的污点分析，来确定code path的走向【不过看作者实验中的描述，貌似是人工指定的路径】。从而才知道要修改哪个函数的参数或结果，以及在何时修改guard value
    - 但总体这个部分写的比较迷，实验也几乎没有提到，似乎主要靠的是约束求解。

Others

最后作者在Related Work中给了非常全面的对比，对于现在的沙箱工具的优劣。虽然这个对比维度，偏向意味很足哈哈哈。但是这个表应该是集齐了现有较好的一些android apk分析框架了。

Summary

总的来说，如果我一开始审到这个稿子，在intro部分可能会眼前一亮，因为确实有这个研究的需求和跨层分析的要求，不过在完整阅读之后，感觉确实都是工程价值罢了，一种炒冷饭的感觉哈哈哈。

不过从阅读流畅度上来说，还是很友好和清晰的。