Introduction and Description of the Work

Android is a popular open-source platform for touchscreen mobile devices, such as smartphones and tablets. Ever since its unveiling in 2007, it has been increasing its market share, running on 68.1% of smartphones sold in the second quarter of 2012 according to a report by the International Data Corporation[1].

Unfortunately, with its growing popularity, the platform became a frequent target of increasingly sophisticated malware, masquerading as legitimate software, while leaking sensitive data about users. More complex malicious applications even try to hide their activity by exploiting security holes of the underlying operating system to bypass the protection mechanisms of the platform.

Researchers have been coming up with different approaches to solving this problem, mostly focusing on porting well-established methods from the desktop environment to resource-limited devices running Android and other competing mobile operating systems. These methods range from static analysis of the executable code, all the way to enforcing sandboxing by virtualization.

One such approach is shown in TaintDroid[2], a project developed by a group of researchers from The Pennsylvania State University, Duke University and Intel Labs. They point out that the access control in Android is rather coarse-grained, with an all-or-nothing policy. This means that once an application is given access to a resource, it can do whatever it wants with it, and does not need any further consent from the user.

TaintDroid modifies several core parts of the Android platform, such as the Dalvik VM, the Android shared library, and even the file-system, adding support for runtime labelling (tainting) of sensitive data like the phone number, the contact list or GPS location, and tracing the flow of such data through the system. By checking the labels of data that are leaving the system, e.g. via the network connection, TaintDroid can warn the users about the possibility of their data being misused. It is therefore an analytical tool providing insight into the behaviour of third-party applications, giving the users a better picture of what happens with the data they entrust to their applications.

Following the work conducted on TaintDroid, the outcome of this project will be a tool with similar goals, but achieving them in a different manner. TaintDroid integrates into the lower levels of the operating system, altering Dalvik's memory management and instructions to store and propagate the taint transparently to the applications running on top of it. However, a similar result can be accomplished by extracting the executable code from the application's package and instrumenting it to carry out the information-flow analysis itself, without any modification to the platform needed, which is what this project will try to attain.

The low-level method makes it possible to trace the tainted data throughout the system

[1] www.idc.com/getdoc.jsp?containerId=prUS23638712

[2] www.appanalysis.org

by patching the IPC kernel module, or by storing the taint within attributes of files. This cannot be done by per-application bytecode instrumentation, but the fact that each application is processed before it is loaded back into the device and executed, leaves room for static analysis of the code, perhaps even modification of its behaviour. Limitations and advantages of both solutions will be thoroughly compared in the evaluation section of the dissertation.

Despite being intended mostly for use by professionals, the configuration of monitored privacy policies should be intuitive enough even for users without deep understanding of dynamic taint analysis. Typical user will:

- connect their Android device to the computer
- choose an installed application that should be instrumented
- select data sources and output channels (sinks) to be monitored
- wait for the application to get instrumented, repackaged and sent back to the device
- run the application and wait for notifications about privacy policy violations

Starting Point

My previous experience includes a UROP internship at the Computer Lab in the summer of 2011. Students worked on security-related projects on the Android platform, an application for encrypted text-messaging in my case.

During my other internship in the summer of 2012, I worked on Java PathFinder, a software-verification tool for JVM. My plugin verified correct usage of physical units in scientific computation by runtime assignment of attributes with unit information to numerical values in the memory, and their propagation during arithmetic operations, which can be regarded as a form of flow tracing.

In this project I hope to combine my prior skills and to apply them in a slightly different environment of the Dalvik VM and Android framework, while learning more about the possibilities of static and dynamic analysis of bytecode.

Substance and Structure of the Project

The main outcome of this project will be a desktop application written in Java, capable of instrumenting given Dalvik executables, known as DEX files, according to options specified via command-line arguments or via graphical user interface.

The development of a dependable code instrumentation, which will reliably identify sources and sinks, and which will propagate the taint throughout the application will form essential part of the project. This requires detailed analysis of the effects of each instruction

supported by the Dalvik VM, and thorough testing. Special attention will need to be given to correct instrumentation of the exception mechanism.

Data sources and sinks will be specified as an XML file containing a list of classes and/or methods inside the Android library, accessing which should trigger the tainting mechanism. The same will apply to so-called sanitizing methods, calling which should not propagate the taint. These are, for example, the hashing functions. Specifications of commonly used sources, sinks and sanitizers will be part of the distribution.

Typical overhead of dynamic data-flow analysis by instruction-level instrumentation varies between 3 and 35 times of the original speed. While there is very limited room for making the instrumentation itself more efficient, time can easily be invested into static analysis of the code which will identify the parts of the code that do not need to be instrumented, i.e. code outside all paths from sources to sinks.

It will also be possible to extract executable code from applications installed on a connected device by executing tools from the Android SDK. After the instrumentation, the modified application will be repackaged and uploaded back to the device. Since all packages need to be signed and there is no access to the original key, packages will be signed by a key of user's choice.

The development process will follow the philosophy of Test Driven Development. Thus a significant amount of time will be spent on building a robust set of unit tests to ensure high quality of code. JUnit will be used to test the internals of the instrumenting application, and a set of shell scripts will be created to automatically execute a collection of tailored snippets of code on an actual Dalvik virtual machine, comparing their output with the output of their instrumented counterparts.

Implicit-flow analysis

TaintDroid focuses strictly on tracing the explicit flow of information, by propagating the taint when data-handling CPU instructions are used. But information can leak via implicit flow (conditional branching instructions) as well. Consider the following example:

```
int sensitiveData = getSensitiveData(); // tainted variable
int leakedData = 0; // taintless variable

while (sensitiveData--)
    leakedData++;

output(leakedData); // leakedData still not tainted
```

In this case, `leakedData` has not been tainted because there is no explicit information flow from `sensitiveData` into it. Instead, `sensitiveData` effects the control flow of the program, leaking the information into `leakedData` implicitly.

Leakage via implicit flow is best identified by static analysis of the source code of the program, a technique often used to analyse scripting languages. Since the source code is not available for third-party applications on Android, it needs to be analysed dynamically on the instruction level. The downside of this solution is that the propagation rules must be rather conservative, leading to false positives. This is why TaintDroid developers decided not to include it into their system-wide solution. Since this project will instrument applications individually, the user will be given the choice of instrumentation including or excluding implicit-flow analysis.

Extensions

Depending on the amount of time necessary to finish the core of the project, extra effort will be put into extending the list of supported features. Each of the following extensions can be implemented separately and they are sorted according to the added benefit to the users in descending order.

Causality analysis

While TaintDroid quite reliably identifies privacy policy violations, it fails to provide information about their origin. The user might therefore have trouble distinguishing whether a violation happened in a background-running thread or as a result of interaction with the UI. By instrumenting the message loop of classes inheriting from the Android Activity class (equivalent of a window), this information could be provided.

Reflection

One major limitation of analysis by bytecode instrumentation is that it cannot easily deal with reflection or dynamically loaded code. The core project will simply warn the user when these are present in the processed code, but a possible extension could further explore methods of handling these.

FastPath optimization

The "A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks" paper, which studied taint-based flow tracing in the context of x86 server applications, suggests a form of dynamic optimization called FastPath. It includes both the original and instrumented version of each method in the resulting executable, and dynamically decides which will be called based on the presence of tainted arguments. Authors of the paper argued that only 2% of typical function calls were tainted, which is why this optimisation had such impact on performance. Whether this is true in the context of Android mobile applications as well is not clear. Before implementing FastPath, a short study of the number of tainted method calls should be conducted.

Success Criteria

For the project to be deemed a success the following items must be successfully completed.

1. Application must communicate with Android devices and be able to download/upload application packages (APKs).
2. Application must be able to extract content of APK files and to repackage them after instrumentation.
3. Specification of sources, sinks and sanitizers has to be designed and these properly identified by the instrumenting application.
4. Instrumentation of DEX files for both explicit and implicit flow analysis needs to be designed, implemented and shown to propagate taint correctly.
5. Capabilities of the information-flow analysis must be demonstrated on examples of real applications.
6. Advantages and limitations of system-level integration versus per-application bytecode instrumentation for the purpose of taint-based flow analysis must be appraised.
7. Performance must be compared to TaintDroid and overhead measured against the original code.

Plan of work

- **Fortnight 1 (2/2 October 2012)**:
  Processing of a DEX file, analysis of its content, saving the content to another file.
- **Fortnight 2 (1/2 November 2012)**:
  Instrumentation of trivial instructions for explicit-flow analysis, identification of sources, sinks and sanitizers.
- **Fortnight 3 (2/2 November 2012)**:
  Communication with Android devices, APK repackaging.
- **Fortnight 4 (1/2 December 2012)**:
  Instrumentation of more complex instructions for explicit-flow analysis, testing on real applications.
- **Fortnight 5 (2/2 December 2012)**:
  Progress report, time for course revision.
- **Fortnight 6 (1/2 January 2013)**:
  Implicit-flow analysis instrumentation, assessment of its impact.

- **Fortnight 7 (2/2 January 2013)**:
  Performance optimisation, benchmarking.
- **Fortnight 8 (1/2 February 2013)**:
  Automated testing on a large sample of applications.
- **Fortnights 9 & 10 (2/2 February 2013, 1/2 March 2013)**:
  Extra time in case of development delay, work on extensions otherwise.
- **Fortnights 11 & 12 (2/2 March 2013, 1/2 April 2013)**:
  Dissertation writing up, first draft given to the supervisor and the Director of Studies by the beginning of Easter term.
- **Fortnights 13 & 14 (2/2 April 2013, 1/2 May 2013)**:
  Incorporating feedback into the dissertation.

Resources Declaration

Development will require computers with the Android SDK installed. Testing on a large sample of applications will be executed on a PWF machine, but most of the work will be done on my personal computer (Asus UL20A laptop, Intel Core 2 Duo 1.3GHz CPU, 4GB RAM, 320GB HDD). I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

Source code will be managed by the Git revision control system, with the repository being hosted on GitHub. Storing the source code in the cloud together with the distributed nature of Git should provide good protection against data loss.

Large collection of infected applications have been obtained from the Android Malware Genome Project[3]. Sufficient disc allocation of 3GB will be needed on the PWF to store a snapshot of the repository.

An Android smartphone will be supplied by the supervisor[4] to test the project on. In case of problems, the Android emulator can be used instead.

[3] www.malgenomeproject.org

[4] Dr Alastair Beresford, arb33@cam.ac.uk