TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones

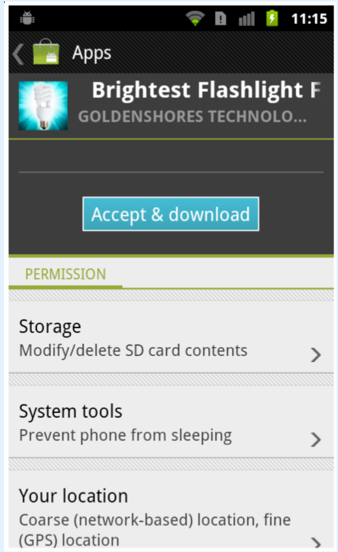
OSDI 2010

William Enck   Peter Gilbert   Byung-Gon Chun
Landon P. Cox   Jaeyeon Jung   Patrick McDaniel
Anmol N. Sheth

November 3, 2015

Presented by Markus

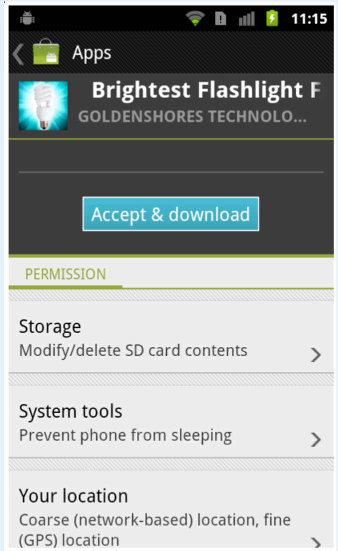
Motivation



- Android permissions are coarse

In general, the privacy permissions on Android applications are fairly coarse grained.
When a user wants to install a third-party application, they must accept the requested permissions of the application
Here we see the ever popular Brightest Flashlight Ever Free
Here, the application requests, amongst other things, the user's location
Now a typical android application is composed of many parts, for example, it may use external advertising libraries
As a result, it can be hard to see which part of the application is using the permission
To continue this example, we don't know if the flashlight is using our GPS to produce light or if some other part of the application is using our data
So, we would like to see if any of our sensitive information, such as our GPS location, can reach the outside world
This is the basic goal of a taint analysis

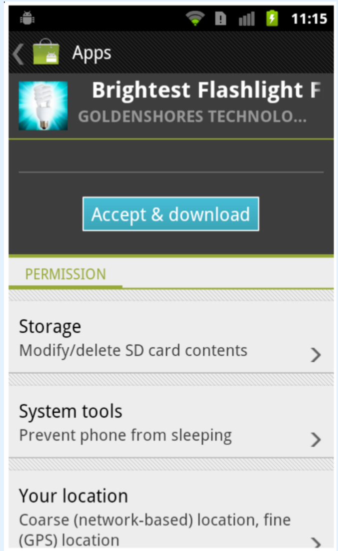
Motivation



- Android permissions are coarse
- Apps are composed of many parts

In general, the privacy permissions on Android applications are fairly coarse grained.
When a user wants to install a third-party application, they must accept the requested permissions of the application
Here we see the ever popular Brightest Flashlight Ever Free
Here, the application requests, amongst other things, the user's location
Now a typical android application is composed of many parts, for example, it may use external advertising libraries
As a result, it can be hard to see which part of the application is using the permission
To continue this example, we don't know if the flashlight is using our GPS to produce light or if some other part of the application is using our data
So, we would like to see if any of our sensitive information, such as our GPS location, can reach the outside world
This is the basic goal of a taint analysis

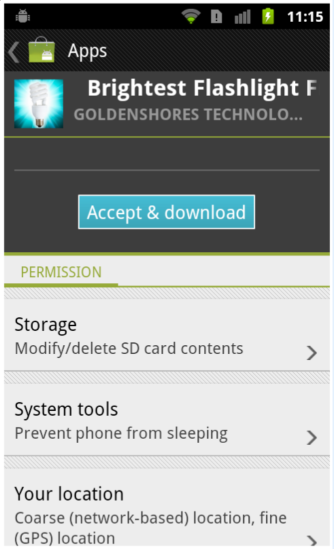
Motivation



- Android permissions are coarse
- Apps are composed of many parts
- "Who is using my GPS data?"

In general, the privacy permissions on Android applications are fairly coarse grained.
When a user wants to install a third-party application, they must accept the requested permissions of the application
Here we see the ever popular Brightest Flashlight Ever Free
Here, the application requests, amongst other things, the user's location
Now a typical android application is composed of many parts, for example, it may use external advertising libraries
As a result, it can be hard to see which part of the application is using the permission
To continue this example, we don't know if the flashlight is using our GPS to produce light or if some other part of the application is using our data
So, we would like to see if any of our sensitive information, such as our GPS location, can reach the outside world
This is the basic goal of a taint analysis

Motivation

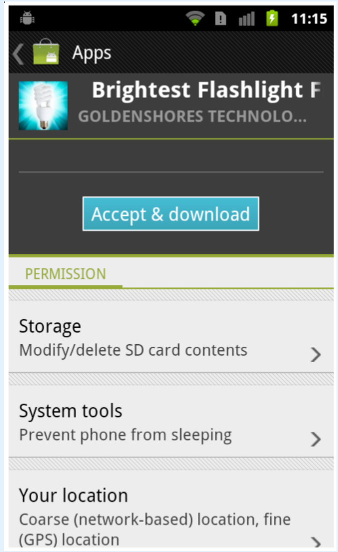


- Android permissions are coarse
- Apps are composed of many parts
- “Who is using my GPS data?”
- Can sensitive data reach the outside world?

In general, the privacy permissions on Android applications are fairly coarse grained.
When a user wants to install a third-party application, they must accept the requested permissions of the application
Here we see the ever popular Brightest Flashlight Ever Free
Here, the application requests, amongst other things, the user's location
Now a typical android application is composed of many parts, for example, it may use external advertising libraries
As a result, it can be hard to see which part of the application is using the permission
To continue this example, we don't know if the flashlight is using our GPS to produce light or if some other part of the application is using our data
So, we would like to see if any of our sensitive information, such as our GPS location, can reach the outside world
This is the basic goal of a taint analysis

Motivation

```
...
double loc = getGPS();
...
double loc2 = loc + 10;
...
if (c) {
  sendGPS(loc2, CIA);
}
```

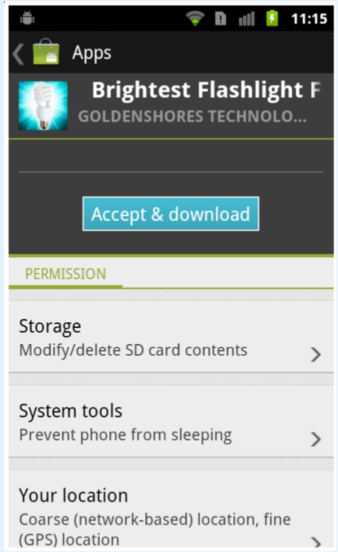


More specifically, we can see how a taint analysis works in practice.
We consider locations where sensitive data is accessed to be taint sources
Locations where data could escape the program are considered as taint sinks
We would like to see if data from a source could be used in a sink
One way to do this is to label sensitive data with a tag
Then, data using labeled data be comes tainted
For example, since loc2 uses loc, it becomes transitively tainted
Then, at a taint sink, you simply check if any of the data is tagged.
In this case, the data is tainted and we could raise a flag to the user.

Motivation

taint source

```
...
double loc = getGPS();
...
double loc2 = loc + 10;
...
if (c) {
  sendGPS(loc2, CIA);
}
```



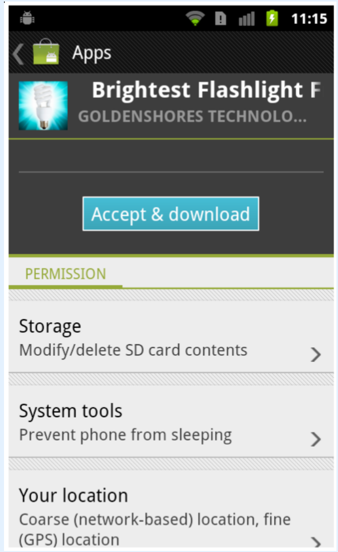
More specifically, we can see how a taint analysis works in practice.
We consider locations where sensitive data is accessed to be taint sources
Locations where data could escape the program are considered as taint sinks
We would like to see if data from a source could be used in a sink
One way to do this is to label sensitive data with a tag
Then, data using labeled data be comes tainted
For example, since loc2 uses loc, it becomes transitively tainted
Then, at a taint sink, you simply check if any of the data is tagged.
In this case, the data is tainted and we could raise a flag to the user.

Motivation

taint source

```
...
double loc = getGPS();
...
double loc2 = loc + 10;
...
if (c) {
  sendGPS(loc2, CIA);
}
```

taint sink



More specifically, we can see how a taint analysis works in practice.
We consider locations where sensitive data is accessed to be taint sources
Locations where data could escape the program are considered as taint sinks
We would like to see if data from a source could be used in a sink
One way to do this is to label sensitive data with a tag
Then, data using labeled data be comes tainted
For example, since loc2 uses loc, it becomes transitively tainted
Then, at a taint sink, you simply check if any of the data is tagged.
In this case, the data is tainted and we could raise a flag to the user.

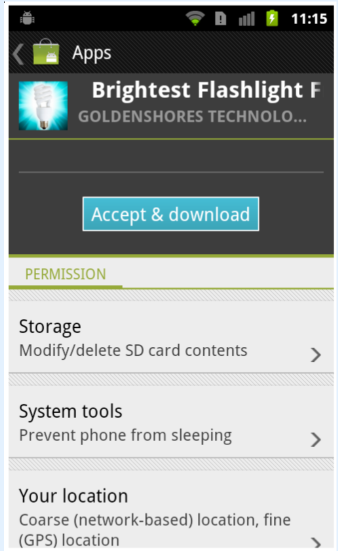
Motivation

taint source

```
...
double loc = getGPS();
...
double loc2 = loc + 10;
...
if (c) {
  sendGPS(loc2, CIA);
}
```

taint sink

Can data from the source be used in the sink?



More specifically, we can see how a taint analysis works in practice.
We consider locations where sensitive data is accessed to be taint sources
Locations where data could escape the program are considered as taint sinks
We would like to see if data from a source could be used in a sink
One way to do this is to label sensitive data with a tag
Then, data using labeled data be comes tainted
For example, since loc2 uses loc, it becomes transitively tainted
Then, at a taint sink, you simply check if any of the data is tagged.
In this case, the data is tainted and we could raise a flag to the user.

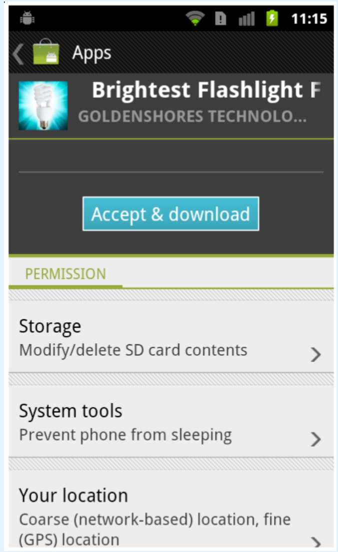
Motivation

taint source

```
...
double loc = getGPS();
...
double loc2 = loc + 10;
...
if (c) {
  sendGPS(loc2, CIA);
}
```

transitively tainted

taint sink



Can data from the source be used in the sink?

More specifically, we can see how a taint analysis works in practice.
We consider locations where sensitive data is accessed to be taint sources
Locations where data could escape the program are considered as taint sinks
We would like to see if data from a source could be used in a sink
One way to do this is to label sensitive data with a tag
Then, data using labeled data be comes tainted
For example, since loc2 uses loc, it becomes transitively tainted
Then, at a taint sink, you simply check if any of the data is tagged.
In this case, the data is tainted and we could raise a flag to the user.

Motivation

App 1

```
double loc = getGPS();
...
...
double loc2 = loc + 10;
...
...
sendMsg(app2, loc2);
```

App 2

```
double recv = recvMsg()
...
...
double t2 = recv - 10;
...
...
socketSend(t2);
```


Here is a more realistic example showing some features of taint droid
Two applications are running and communicating to each other using message passing
As we saw last week, it is possible for an application with, for example, GPS access permissions to send GPS data to an application without GPS permissions
Taint droid is capable of tracking this
First, an application reads GPS data
It is then used in some intermediate computations before being sent in a message
The tainted data then travels through the operating system where it is recieved in another process
In the other process, the tainted data is used again in some computations before it is finally sent out over the network through a socket
TaintDroid is able to track the taint information both inside the processses and through android's message passing framework to detect possible leaks of sensitive information through the network

Motivation

App 1

taint source

```
double loc = getGPS();
...
...
double loc2 = loc + 10;
...
...
sendMsg(app2, loc2);
```

App 2

```
double recv = recvMsg()
...
...
double t2 = recv - 10;
...
...
socketSend(t2);
```


Here is a more realistic example showing some features of taint droid
Two applications are running and communicating to each other using message passing
As we saw last week, it is possible for an application with, for example, GPS access permissions to send GPS data to an application without GPS permissions
Taint droid is capable of tracking this
First, an application reads GPS data
It is then used in some intermediate computations before being sent in a message
The tainted data then travels through the operating system where it is recieved in another process
In the other process, the tainted data is used again in some computations before it is finally sent out over the network through a socket
TaintDroid is able to track the taint information both inside the processses and through android's message passing framework to detect possible leaks of sensitive information through the network

Motivation

App 1

```
double loc = getGPS();
...
...
double loc2 = loc + 10;
...
...
sendMsg(app2, loc2);
```

taint source

transitively tainted

App 2

```
double recv = recvMsg()
...
...
double t2 = recv - 10;
...
...
socketSend(t2);
```


Here is a more realistic example showing some features of taint droid
Two applications are running and communicating to each other using message passing
As we saw last week, it is possible for an application with, for example, GPS access permissions to send GPS data to an application without GPS permissions
Taint droid is capable of tracking this
First, an application reads GPS data
It is then used in some intermediate computations before being sent in a message
The tainted data then travels through the operating system where it is recieved in another process
In the other process, the tainted data is used again in some computations before it is finally sent out over the network through a socket
TaintDroid is able to track the taint information both inside the processses and through android's message passing framework to detect possible leaks of sensitive information through the network

Motivation

App 1

```
double loc = getGPS();
...
...
double loc2 = loc + 10;
...
...
sendMsg(app2, loc2);
```

taint source

transitively tainted

tainted msg send

App 2

```
double recv = recvMsg()
...
...
double t2 = recv - 10;
...
...
socketSend(t2);
```


Here is a more realistic example showing some features of taint droid
Two applications are running and communicating to each other using message passing
As we saw last week, it is possible for an application with, for example, GPS access permissions to send GPS data to an application without GPS permissions
Taint droid is capable of tracking this
First, an application reads GPS data
It is then used in some intermediate computations before being sent in a message
The tainted data then travels through the operating system where it is recieved in another process
In the other process, the tainted data is used again in some computations before it is finally sent out over the network through a socket
TaintDroid is able to track the taint information both inside the processses and through android's message passing framework to detect possible leaks of sensitive information through the network

Motivation

App 1

```
double loc = getGPS();
...
...
double loc2 = loc + 10;
...
...
sendMsg(app2, loc2);
```

taint source

transitively tainted

tainted msg send

App 2

```
double recv = recvMsg()
...
...
double t2 = recv - 10;
...
...
socketSend(t2);
```

tainted msg recv

Here is a more realistic example showing some features of taint droid
Two applications are running and communicating to each other using message passing
As we saw last week, it is possible for an application with, for example, GPS access permissions to send GPS data to an application without GPS permissions
Taint droid is capable of tracking this
First, an application reads GPS data
It is then used in some intermediate computations before being sent in a message
The tainted data then travels through the operating system where it is recieved in another process
In the other process, the tainted data is used again in some computations before it is finally sent out over the network through a socket
TaintDroid is able to track the taint information both inside the processses and through android's message passing framework to detect possible leaks of sensitive information through the network

Motivation

App 1

```
double loc = getGPS();      ← taint source
...
...
double loc2 = loc + 10;     ← transitively tainted
...
...
sendMsg(app2, loc2);        ← tainted msg send
```

App 2

```
double recv = recvMsg()     ← tainted msg recv
...
...
double t2 = recv - 10;      ← transitively tainted
...
...
socketSend(t2);             ← tainted network send
```


Here is a more realistic example showing some features of taint droid

Two applications are running and communicating to each other using message passing

As we saw last week, it is possible for an application with, for example, GPS access permissions to send GPS data to an application without GPS permissions

Taint droid is capable of tracking this

First, an application reads GPS data

It is then used in some intermediate computations before being sent in a message

The tainted data then travels through the operating system where it is recieved in another process

In the other process, the tainted data is used again in some computations before it is finally sent out over the network through a socket

TaintDroid is able to track the taint information both inside the processses and through android's message passing framework to detect possible leaks of sensitive information through the network

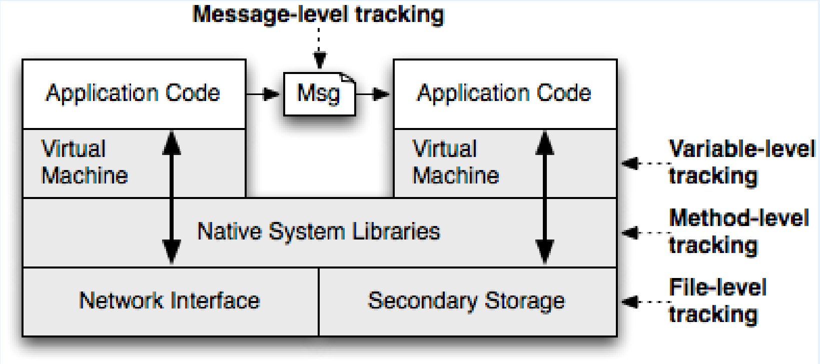
TaintDroid

- TaintDroid: Dynamic Taint Analysis for Android
- Detects transmission of sensitive data
- Low runtime overhead (≈14%)
- Does not require source code
- Implemented ontop of Dalvik VM

androidcentral.com

This brings us to the authors contribution
They present TaintDroid, a dynamic taint analyzer for Android Applications
As presented in the previous example, the goal is to detect the transmission of sensitive data
Since their analysis is dynamic, it is performed while the application is run.
The authors note that there is a minimal overhead and little precieved latency in the applications
Additionaly, since they modified the Dalvik VM, their method does not require the source code of the program.

How?



This figure shows a high-level overview of their approach
First, all variables reading from taint sources are labeled
The Dalvik VM is modified in order to track the propegation of the label between variables
Since applications can communicate, they also track taint flows through interprocess communcation
As we saw last week, data from trusted applications can flow to untrusted applications due to insecure IPC. TaintDroid is able to detect this.
Because they use the VM to propegate taint information, they may lose data through native function calls
So, they monitor executed methods to propegate taint information through known native code
Finally, they track tainted data through files
By monitoring the network interface, they can see if tainted data can escape
Since they are storing and tracking information, a big part of this work, as we will see, is a tradeoff between acurracy and scalability

Overview

Next, I will go over some background information about android

Android Components

- Dalvik VM interpreter
- Native code
- Binder IPC

xda-developers.com

In order to understand their work, we need to briefly go over three parts of the Android OS
First, we will talk about the Dalvik VM Intepreter, Android's Java VM.
Then, we will discuss Java applications calling into native code
And finally, we will look at the binder message passing system

Dalvik VM

- Dalvik EXecutable (DEX) byte code

Android applications are most commonly written in Java and then are compiled to Dalvik Executables
Each individual application has its own Dalvik interpreter instance
The DEX language itself is register based (as opposed to stack based)
All operations are performed on registers; values must first be loaded and then afterwards stored
Looks a lot like other register machines
As we will see later, the structure of the language determines how taint information is propagated

Dalvik VM

- Dalvik EXecutable (DEX) byte code
- Each application has a Dalvik interpreter instance

Android applications are most commonly written in Java and then are compiled to Dalvik Executables
Each individual application has its own Dalvik interpreter instance
The DEX language itself is register based (as opposed to stack based)
All operations are performed on registers; values must first be loaded and then afterwards stored
Looks a lot like other register machines
As we will see later, the structure of the language determines how taint information is propagated

Dalvik VM

- Dalvik EXecutable (DEX) byte code
- Each application has a Dalvik interpreter instance
- DEX is a register based language

Android applications are most commonly written in Java and then are compiled to Dalvik Executables
Each individual application has its own Dalvik interpreter instance
The DEX language itself is register based (as opposed to stack based)
All operations are performed on registers; values must first be loaded and then afterwards stored
Looks a lot like other register machines
As we will see later, the structure of the language determines how taint information is propagated

Dalvik VM

- Dalvik EXecutable (DEX) byte code
- Each application has a Dalvik interpreter instance
- DEX is a register based language
  - `move vA, vB`

Android applications are most commonly written in Java and then are compiled to Dalvik Executables
Each individual application has its own Dalvik interpreter instance
The DEX language itself is register based (as opposed to stack based)
All operations are performed on registers; values must first be loaded and then afterwards stored
Looks a lot like other register machines
As we will see later, the structure of the language determines how taint information is propagated

Dalvik VM

- Dalvik EXecutable (DEX) byte code
- Each application has a Dalvik interpreter instance
- DEX is a register based language
  - `move vA, vB`
  - `add-int dst, src1, src2`

Android applications are most commonly written in Java and then are compiled to Dalvik Executables
Each individual application has its own Dalvik interpreter instance
The DEX language itself is register based (as opposed to stack based)
All operations are performed on registers; values must first be loaded and then afterwards stored
Looks a lot like other register machines
As we will see later, the structure of the language determines how taint information is propagated

Dalvik VM

- Dalvik EXecutable (DEX) byte code
- Each application has a Dalvik interpreter instance
- DEX is a register based language
  - `move vA, vB`
  - `add-int dst, src1, src2`
- Language structure determines propagation rules

Android applications are most commonly written in Java and then are compiled to Dalvik Executables
Each individual application has its own Dalvik interpreter instance
The DEX language itself is register based (as opposed to stack based)
All operations are performed on registers; values must first be loaded and then afterwards stored
Looks a lot like other register machines
As we will see later, the structure of the language determines how taint information is propagated

Native Code

- Android allows Java to execute native code

Android allows applications to call into native code
This can be used, for example, to optimize performance, access third-party libraries like OpenGL, or call into kernel functions
From a security standpoint, one key feature of native code execution is that it has access to Java internals
So, since the authors are taint tracking on the Java VM, the native code is not tracked and must be trusted
To track taint flow through trusted native code, the authors use the semantics of the native functions and the taint information of the arguments
For example, if there is a native method to write data to a file, the authors know that if the argument is tainted the file becomes tainted.

Native Code

- Android allows Java to execute native code
  - Performance optimizations

Android allows applications to call into native code
This can be used, for example, to optimize performance, access third-party libraries like OpenGL, or call into kernel functions
From a security standpoint, one key feature of native code execution is that it has access to Java internals
So, since the authors are taint tracking on the Java VM, the native code is not tracked and must be trusted
To track taint flow through trusted native code, the authors use the semantics of the native functions and the taint information of the arguments
For example, if there is a native method to write data to a file, the authors know that if the argument is tainted the file becomes tainted.

Native Code

- Android allows Java to execute native code
  - Performance optimizations
  - Third-party libraries (OpenGL, WebKit)

Android allows applications to call into native code
This can be used, for example, to optimize performance, access third-party libraries like OpenGL, or call into kernel functions
From a security standpoint, one key feature of native code execution is that it has access to Java internals
So, since the authors are taint tracking on the Java VM, the native code is not tracked and must be trusted
To track taint flow through trusted native code, the authors use the semantics of the native functions and the taint information of the arguments
For example, if there is a native method to write data to a file, the authors know that if the argument is tainted the file becomes tainted.

Native Code

- Android allows Java to execute native code
  - Performance optimizations
  - Third-party libraries (OpenGL, WebKit)
  - Kernel functions

Android allows applications to call into native code
This can be used, for example, to optimize performance, access third-party libraries like OpenGL, or call into kernel functions
From a security standpoint, one key feature of native code execution is that it has access to Java internals
So, since the authors are taint tracking on the Java VM, the native code is not tracked and must be trusted
To track taint flow through trusted native code, the authors use the semantics of the native functions and the taint information of the arguments
For example, if there is a native method to write data to a file, the authors know that if the argument is tainted the file becomes tainted.

Native Code

- Android allows Java to execute native code
  - Performance optimizations
  - Third-party libraries (OpenGL, WebKit)
  - Kernel functions
- Can access Java internals

Android allows applications to call into native code
This can be used, for example, to optimize performance, access third-party libraries like OpenGL, or call into kernel functions
From a security standpoint, one key feature of native code execution is that it has access to Java internals
So, since the authors are taint tracking on the Java VM, the native code is not tracked and must be trusted
To track taint flow through trusted native code, the authors use the semantics of the native functions and the taint information of the arguments
For example, if there is a native method to write data to a file, the authors know that if the argument is tainted the file becomes tainted.

Native Code

- Android allows Java to execute native code
  - Performance optimizations
  - Third-party libraries (OpenGL, WebKit)
  - Kernel functions
- Can access Java internals
  - Native code must be trusted

Android allows applications to call into native code
This can be used, for example, to optimize performance, access third-party libraries like OpenGL, or call into kernel functions
From a security standpoint, one key feature of native code execution is that it has access to Java internals
So, since the authors are taint tracking on the Java VM, the native code is not tracked and must be trusted
To track taint flow through trusted native code, the authors use the semantics of the native functions and the taint information of the arguments
For example, if there is a native method to write data to a file, the authors know that if the argument is tainted the file becomes tainted.

Native Code

- Android allows Java to execute native code
  - Performance optimizations
  - Third-party libraries (OpenGL, WebKit)
  - Kernel functions
- Can access Java internals
  - Native code must be trusted
- Model taint propagation through native code

Android allows applications to call into native code
This can be used, for example, to optimize performance, access third-party libraries like OpenGL, or call into kernel functions
From a security standpoint, one key feature of native code execution is that it has access to Java internals
So, since the authors are taint tracking on the Java VM, the native code is not tracked and must be trusted
To track taint flow through trusted native code, the authors use the semantics of the native functions and the taint information of the arguments
For example, if there is a native method to write data to a file, the authors know that if the argument is tainted the file becomes tainted.

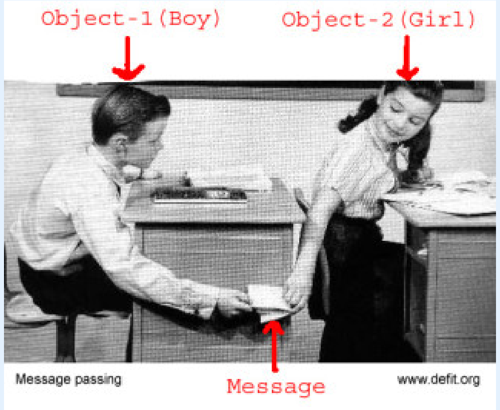
Native Code

- Android allows Java to execute native code
  - Performance optimizations
  - Third-party libraries (OpenGL, WebKit)
  - Kernel functions
- Can access Java internals
  - Native code must be trusted
- Model taint propagation through native code
  - `writeFile(taint)`

Android allows applications to call into native code
This can be used, for example, to optimize performance, access third-party libraries like OpenGL, or call into kernel functions
From a security standpoint, one key feature of native code execution is that it has access to Java internals
So, since the authors are taint tracking on the Java VM, the native code is not tracked and must be trusted
To track taint flow through trusted native code, the authors use the semantics of the native functions and the taint information of the arguments
For example, if there is a native method to write data to a file, the authors know that if the argument is tainted the file becomes tainted.

Binder IPC

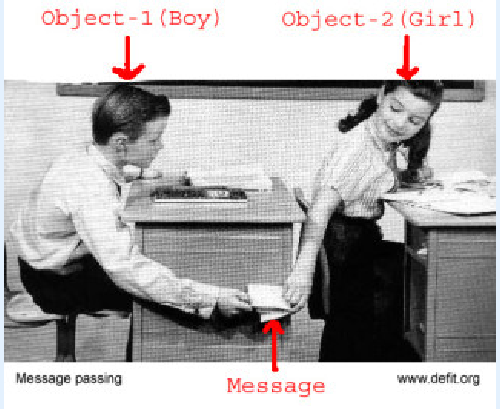
- Inter-process communication goes through Binder



Android IPC uses a framework called binder.
Processes define an interface allowing them to accept data
For the sake of understanding this work, we can use a simplistic view of how IPC works
First, a process bundles up a bunch of data to send
Then it performs the send operations
And finally, the receiving process unpacks the data
Hopefully we can see how this pertains to taint tracking: tainted data in a message results in a tainted read

Binder IPC

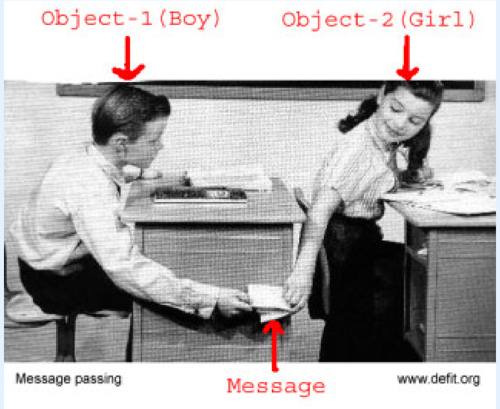
- Inter-process communication goes through Binder
- Messages sent via defined interfaces



Android IPC uses a framework called binder.
Processes define an interface allowing them to accept data
For the sake of understanding this work, we can use a simplistic view of how IPC works
First, a process bundles up a bunch of data to send
Then it performs the send operations
And finally, the receiving process unpacks the data
Hopefully we can see how this pertains to taint tracking: tainted data in a message results in a tainted read

Binder IPC

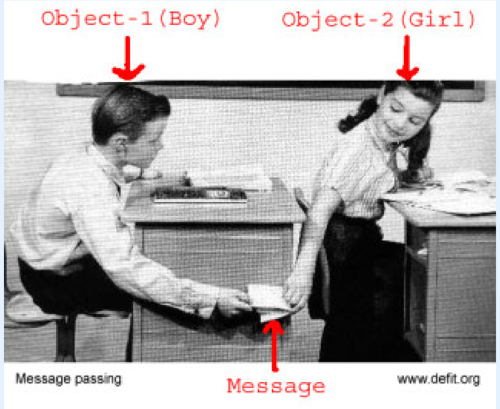
- Inter-process communication goes through Binder
- Messages sent via defined interfaces
- Process A parcels data



Android IPC uses a framework called binder.
Processes define an interface allowing them to accept data
For the sake of understanding this work, we can use a simplistic view of how IPC works
First, a process bundles up a bunch of data to send
Then it performs the send operations
And finally, the receiving process unpacks the data
Hopefully we can see how this pertains to taint tracking: tainted data in a message results in a tainted read

Binder IPC

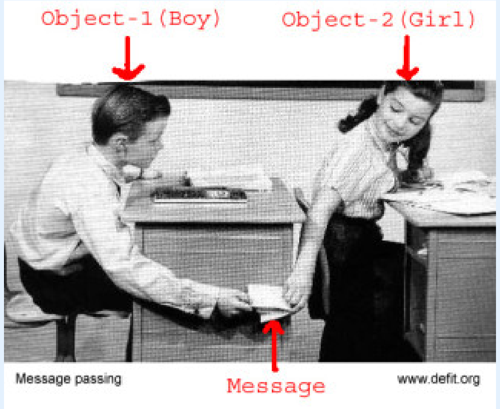
- Inter-process communication goes through Binder
- Messages sent via defined interfaces
- Process A parcels data
- Process A send data to Process B



Android IPC uses a framework called binder.
Processes define an interface allowing them to accept data
For the sake of understanding this work, we can use a simplistic view of how IPC works
First, a process bundles up a bunch of data to send
Then it performs the send operations
And finally, the receiving process unpacks the data
Hopefully we can see how this pertains to taint tracking: tainted data in a message results in a tainted read

Binder IPC

- Inter-process communication goes through Binder
- Messages sent via defined interfaces
- Process A parcels data
- Process A send data to Process B
- Process B reads parcel



Android IPC uses a framework called binder.
Processes define an interface allowing them to accept data
For the sake of understanding this work, we can use a simplistic view of how IPC works
First, a process bundles up a bunch of data to send
Then it performs the send operations
And finally, the receiving process unpacks the data
Hopefully we can see how this pertains to taint tracking: tainted data in a message results in a tainted read

Overview

Introduction

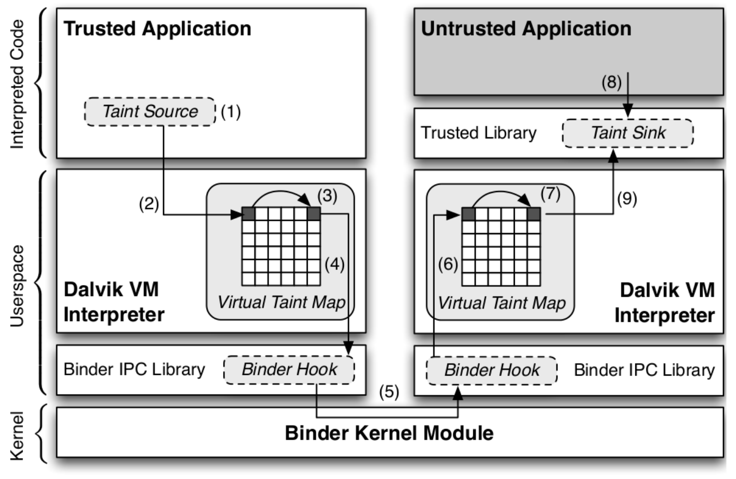
Background

TaintDroid
- Example
- Interpreted Code
- Native Code
- IPC
- Files

Experiments

Next, I'll go over more of the details of how TaintDroid is setup
To do this I'll first present an example showing all the features of TaintDroid
Then, I'll discuss taint tracking in the VM, through native code, and through IPC

Example



Here is a scenario showing the features of TaintDroid
First, a trusted application reads some secure data, for example, the user's location
Taint information is stored in what the authors call TaintTags
Then, the trusted application uses the tainted data in some VM operations
Through the operations, such as addition or subtraction, or through native methods, the taint data propagates
Then, when the tainted application uses IPC, the kernel Binder module captures the taint information at the send point
The parcel is passed through the kernel to the receiving application
When the parcel is unpacked, the receiving process has the taint information from the sender
Then, the receiving application uses the tainted data within its VM, thereby propagating the taint information
Finally, the receiving application runs a taint sink, for example a network send, with the tainted IPC data raising an alarm
To track all this taint information, TaintDroid must handle interpreted code, native code, and IPC.

Taint Tags

- Associate taint tag with each variable

To track propagation of taint information, each variable in the application is associated with a taint tag
The size of the taint tag, or the granularity at which variables are monitored has a big influence on performance
TaintDroid assigns, to each monitored variable, a 32-bit bit-vector
The bit-vector is adjacent to the variable to make use of spatial locality
32-bits allow for the user to 32 different taint markings, for example, allowing different taint sources to be tracked independently
To track local variables, which are stored on a stack similar to x86, the stack allocation size is double and each variable gets an extra taint bitvector
To handle the performance overhead, an array variable has only one 32-bit tag.
Because of this, there is an increase in performance and false positives

Taint Tags

- Associate taint tag with each variable
  - Performance/Memory overhead

To track propagation of taint information, each variable in the application is associated with a taint tag
The size of the taint tag, or the granularity at which variables are monitored has a big influence on performance
TaintDroid assigns, to each monitored variable, a 32-bit bit-vector
The bit-vector is adjacent to the variable to make use of spatial locality
32-bits allow for the user to 32 different taint markings, for example, allowing different taint sources to be tracked independently
To track local variables, which are stored on a stack similar to x86, the stack allocation size is double and each variable gets an extra taint bitvector
To handle the performance overhead, an array variable has only one 32-bit tag.
Because of this, there is an increase in performance and false positives

Taint Tags

- Associate taint tag with each variable
  - Performance/Memory overhead
- Monitored variables have an adjacent 32-bit bit-vector

To track propagation of taint information, each variable in the application is associated with a taint tag
The size of the taint tag, or the granularity at which variables are monitored has a big influence on performance
TaintDroid assigns, to each monitored variable, a 32-bit bit-vector
The bit-vector is adjacent to the variable to make use of spatial locality
32-bits allow for the user to 32 different taint markings, for example, allowing different taint sources to be tracked independently
To track local variables, which are stored on a stack similar to x86, the stack allocation size is double and each variable gets an extra taint bitvector
To handle the performance overhead, an array variable has only one 32-bit tag.
Because of this, there is an increase in performance and false positives

Taint Tags

- Associate taint tag with each variable
  - Performance/Memory overhead
- Monitored variables have an adjacent 32-bit bit-vector
- Local variables stored on stack

To track propagation of taint information, each variable in the application is associated with a taint tag
The size of the taint tag, or the granularity at which variables are monitored has a big influence on performance
TaintDroid assigns, to each monitored variable, a 32-bit bit-vector
The bit-vector is adjacent to the variable to make use of spatial locality
32-bits allow for the user to 32 different taint markings, for example, allowing different taint sources to be tracked independently
To track local variables, which are stored on a stack similar to x86, the stack allocation size is double and each variable gets an extra taint bitvector
To handle the performance overhead, an array variable has only one 32-bit tag.
Because of this, there is an increase in performance and false positives

Taint Tags

- Associate taint tag with each variable
  - Performance/Memory overhead
- Monitored variables have an adjacent 32-bit bit-vector
- Local variables stored on stack
- Double stack frame size for taint tags

To track propagation of taint information, each variable in the application is associated with a taint tag
The size of the taint tag, or the granularity at which variables are monitored has a big influence on performance
TaintDroid assigns, to each monitored variable, a 32-bit bit-vector
The bit-vector is adjacent to the variable to make use of spatial locality
32-bits allow for the user to 32 different taint markings, for example, allowing different taint sources to be tracked independently
To track local variables, which are stored on a stack similar to x86, the stack allocation size is double and each variable gets an extra taint bitvector
To handle the performance overhead, an array variable has only one 32-bit tag.
Because of this, there is an increase in performance and false positives

Taint Tags

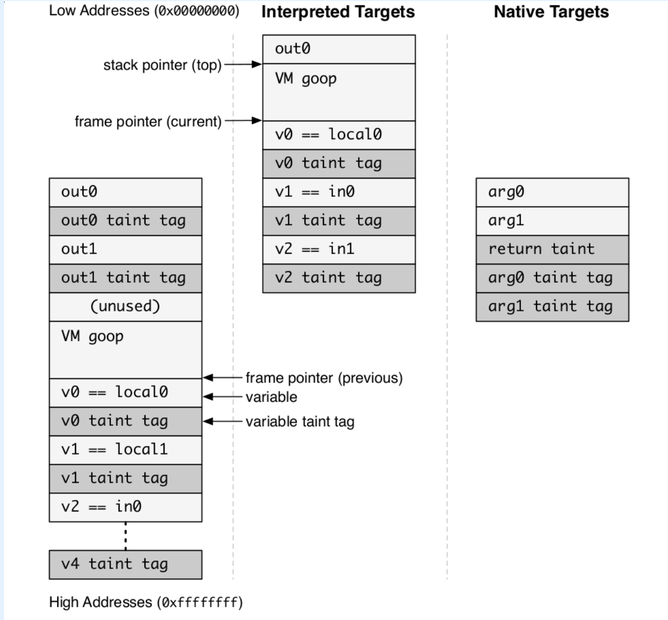
- Associate taint tag with each variable
  - Performance/Memory overhead
- Monitored variables have an adjacent 32-bit bit-vector
- Local variables stored on stack
- Double stack frame size for taint tags
- One 32-bit tag per array

To track propagation of taint information, each variable in the application is associated with a taint tag
The size of the taint tag, or the granularity at which variables are monitored has a big influence on performance
TaintDroid assigns, to each monitored variable, a 32-bit bit-vector
The bit-vector is adjacent to the variable to make use of spatial locality
32-bits allow for the user to 32 different taint markings, for example, allowing different taint sources to be tracked independently
To track local variables, which are stored on a stack similar to x86, the stack allocation size is double and each variable gets an extra taint bitvector
To handle the performance overhead, an array variable has only one 32-bit tag.
Because of this, there is an increase in performance and false positives

Taint Tag Example

- 32-bit tags allow multiple sources to be tracked
  - Bit 1: GPS Location
  - Bit 2: Phone number
  - ...

The use of the 32-bit taint tags is left up to the user
Each bit could be used to track a single taint source
This allows the user to see not just if tainted information flows into a sink but also what type of taint source flows into a sink



On the left hand side of this image we can see the modified stack to include taint bitvectors
The dark gray items are the added taint tags and the light gray items are the variables in the application
You can see how the taint tags are interleaved with all the variables

Array Inaccuracies

```
arr[1] = notTainted();
arr[0] = taint();
t1 = arr[1]; // tainted!
taintSink(t1); // false
               // alarm
```

- Over-approximate array with a single taint tag
- Considers all array elements as single element
- False alarms

By considering array's as a single element the authors gain performance but increase the amount of false alarms
If a tainted item is storred into an array then the entire array becomes tainted
As a result, a read of any value from the array leads to taint propagation
In other words, all items in an array are considered to be a single element
In this example, we see index 1 of the array is not tainted while index 0 is.
A subsequent read of index 1, even though in reality it is not tainted, results in taint propagation from index 0 to 1
So, if t1 is passed to a taintSink, a false alarm will be generated.

Interpreted Code Taint Propagation

- Variable level taint tracking
- Based on structural semantics of DEX code
- Tracks through primitive types and object references

As said before, inside the Dalvik VM there is variable level taint tracking
The taint propagation, as I will show soon, is based on the structural semantics of the DEX machine language
This is essentially the same as the static and dynamic taint analyses we've been where each statement in the program has associated semantics releative to the analysis

Taint Propagation Logic

- $\mathcal{L}$: all possible taint markings

First, we define the set of all possible taint markings to be L
A taint tag for a variable is one of these possible taint markings
Local and argument variables are stored on registers denoted by v.
Fields of a class are represented through f
Static fields can directly be accessed through the class
For a given instance of a class, the instance variable is accessed with the register and field name
This is done similar to the familiar dot notation in an actual programming language
And finally, array accesses use the typical square bracket notation
The taint map is defined using tau. It associates with each variable a taint tag
The arrow allows for updates and retrievals of a variables taint information.
We can read the taint information of a variable or update the taint information.

Taint Propagation Logic

- $\mathcal{L}$: all possible taint markings
- $t \in \mathcal{L}$: taint tag

First, we define the set of all possible taint markings to be L
A taint tag for a variable is one of these possible taint markings
Local and argument variables are stored on registers denoted by v.
Fields of a class are represented through f
Static fields can directly be accessed through the class
For a given instance of a class, the instance variable is accessed with the register and field name
This is done similar to the familiar dot notation in an actual programming language
And finally, array accesses use the typical square bracket notation
The taint map is defined using tau. It associates with each variable a taint tag
The arrow allows for updates and retrievals of a variables taint information.
We can read the taint information of a variable or update the taint information.

Taint Propagation Logic

- $\mathcal{L}$: all possible taint markings
- $t \in \mathcal{L}$: taint tag
- v_x: virtual register x

First, we define the set of all possible taint markings to be L
A taint tag for a variable is one of these possible taint markings
Local and argument variables are stored on registers denoted by v.
Fields of a class are represented through f
Static fields can directly be accessed through the class
For a given instance of a class, the instance variable is accessed with the register and field name
This is done similar to the familiar dot notation in an actual programming language
And finally, array accesses use the typical square bracket notation
The taint map is defined using tau. It associates with each variable a taint tag
The arrow allows for updates and retrievals of a variables taint information.
We can read the taint information of a variable or update the taint information.

Taint Propagation Logic

- $\mathcal{L}$: all possible taint markings
- $t \in \mathcal{L}$: taint tag
- v_x: virtual register x
- f_x: field variable of class x

First, we define the set of all possible taint markings to be L
A taint tag for a variable is one of these possible taint markings
Local and argument variables are stored on registers denoted by v.
Fields of a class are represented through f
Static fields can directly be accessed through the class
For a given instance of a class, the instance variable is accessed with the register and field name
This is done similar to the familiar dot notation in an actual programming language
And finally, array accesses use the typical square bracket notation
The taint map is defined using tau. It associates with each variable a taint tag
The arrow allows for updates and retrievals of a variables taint information.
We can read the taint information of a variable or update the taint information.

Taint Propagation Logic

- $\mathcal{L}$: all possible taint markings
- $t \in \mathcal{L}$: taint tag
- v_x: virtual register x
- f_x: field variable of class x
- f_x: static field

First, we define the set of all possible taint markings to be L
A taint tag for a variable is one of these possible taint markings
Local and argument variables are stored on registers denoted by v.
Fields of a class are represented through f
Static fields can directly be accessed through the class
For a given instance of a class, the instance variable is accessed with the register and field name
This is done similar to the familiar dot notation in an actual programming language
And finally, array accesses use the typical square bracket notation
The taint map is defined using tau. It associates with each variable a taint tag
The arrow allows for updates and retrievals of a variables taint information.
We can read the taint information of a variable or update the taint information.

Taint Propagation Logic

- $\mathcal{L}$: all possible taint markings
- $t \in \mathcal{L}$: taint tag
- v_x: virtual register x
- f_x: field variable of class x
- f_x: static field
- $v_x(f_x)$: instance field (`vx.x`)

First, we define the set of all possible taint markings to be L
A taint tag for a variable is one of these possible taint markings
Local and argument variables are stored on registers denoted by v.
Fields of a class are represented through f
Static fields can directly be accessed through the class
For a given instance of a class, the instance variable is accessed with the register and field name
This is done similar to the familiar dot notation in an actual programming language
And finally, array accesses use the typical square bracket notation
The taint map is defined using tau. It associates with each variable a taint tag
The arrow allows for updates and retrievals of a variables taint information.
We can read the taint information of a variable or update the taint information.

Taint Propagation Logic

- $\mathcal{L}$: all possible taint markings
- $t \in \mathcal{L}$: taint tag
- v_x: virtual register x
- f_x: field variable of class x
- f_x: static field
- $v_x(f_x)$: instance field (`vx.x`)
- $v_x[\cdot]$: array access

First, we define the set of all possible taint markings to be L
A taint tag for a variable is one of these possible taint markings
Local and argument variables are stored on registers denoted by v.
Fields of a class are represented through f
Static fields can directly be accessed through the class
For a given instance of a class, the instance variable is accessed with the register and field name
This is done similar to the familiar dot notation in an actual programming language
And finally, array accesses use the typical square bracket notation
The taint map is defined using tau. It associates with each variable a taint tag
The arrow allows for updates and retrievals of a variables taint information.
We can read the taint information of a variable or update the taint information.

Taint Propagation Logic

- $\mathcal{L}$: all possible taint markings
- $t \in \mathcal{L}$: taint tag
- v_x: virtual register x
- f_x: field variable of class x
- f_x: static field
- $v_x(f_x)$: instance field (`vx.x`)
- $v_x[\cdot]$: array access
- $\tau(v) \leftarrow t$: set taint tag of v to t

First, we define the set of all possible taint markings to be L
A taint tag for a variable is one of these possible taint markings
Local and argument variables are stored on registers denoted by v.
Fields of a class are represented through f
Static fields can directly be accessed through the class
For a given instance of a class, the instance variable is accessed with the register and field name
This is done similar to the familiar dot notation in an actual programming language
And finally, array accesses use the typical square bracket notation
The taint map is defined using tau. It associates with each variable a taint tag
The arrow allows for updates and retrievals of a variables taint information.
We can read the taint information of a variable or update the taint information.

Taint Propagation Logic

- $\mathcal{L}$: all possible taint markings
- $t \in \mathcal{L}$: taint tag
- v_x: virtual register x
- f_x: field variable of class x
- f_x: static field
- $v_x(f_x)$: instance field (`vx.x`)
- $v_x[\cdot]$: array access
- $\tau(v) \leftarrow t$: set taint tag of v to t
- $t \leftarrow \tau(v)$: set t to taint tag of v

First, we define the set of all possible taint markings to be L
A taint tag for a variable is one of these possible taint markings
Local and argument variables are stored on registers denoted by v.
Fields of a class are represented through f
Static fields can directly be accessed through the class
For a given instance of a class, the instance variable is accessed with the register and field name
This is done similar to the familiar dot notation in an actual programming language
And finally, array accesses use the typical square bracket notation
The taint map is defined using tau. It associates with each variable a taint tag
The arrow allows for updates and retrievals of a variables taint information.
We can read the taint information of a variable or update the taint information.

Taint Propagation Logic

DEX Instruction	Semantics	Propagation
move v_a v_b	$v_a \leftarrow v_b$	$\tau(v_a) \leftarrow \tau(v_b)$

Given our new language, we can analyze the structure of each DEX instruction to show how it updates the taint information
We'll look at a few of the instruction and show how this is done
First, for the DEX move instruction, the semantics is like assignment in Java. The value of v_b is assigned to v_a
In terms of the taint update, this simply propagates the taint information from v_b to v_a
A unary operation works similarly: the actual semantics of the operator is ignored and the taint values are propegated
Finally, an array update stores some value v_a into array v_b at location v_c
Here we can see how the taint information of the entire array is stored into one value.
An array update takes the taint information of the entire array and then conjoins it with the taint information of the assigned variable
Operations for the remainder of the DEX language are defined in the paper
What's nice about this approach is we write simple rules for each instruction and by structural induction we get a way to define the taint flow for the entire program

Taint Propagation Logic

DEX Instruction	Semantics	Propagation
move v_a v_b	$v_a \leftarrow v_b$	$\tau(v_a) \leftarrow \tau(v_b)$
unary v_a v_b	$v_a \leftarrow \otimes v_a$	$\tau(v_a) \leftarrow \tau(v_b)$

Given our new language, we can analyze the structure of each DEX instruction to show how it updates the taint information
We'll look at a few of the instruction and show how this is done
First, for the DEX move instruction, the semantics is like assignment in Java. The value of v_b is assigned to v_a
In terms of the taint update, this simply propagates the taint information from v_b to v_a
A unary operation works similarly: the actual semantics of the operator is ignored and the taint values are propegated
Finally, an array update stores some value v_a into array v_b at location v_c
Here we can see how the taint information of the entire array is stored into one value.
An array update takes the taint information of the entire array and then conjoins it with the taint information of the assigned variable
Operations for the remainder of the DEX language are defined in the paper
What's nice about this approach is we write simple rules for each instruction and by structural induction we get a way to define the taint flow for the entire program

Taint Propagation Logic

DEX Instruction	Semantics	Propagation
move v_a v_b	$v_a \leftarrow v_b$	$\tau(v_a) \leftarrow \tau(v_b)$
unary v_a v_b	$v_a \leftarrow \otimes v_a$	$\tau(v_a) \leftarrow \tau(v_b)$
aput v_a v_b v_c	$v_b[v_c] \leftarrow v_a$	$\tau(v_b[\cdot]) \leftarrow \tau(v_b[\cdot]) \cup \tau(v_a)$

Given our new language, we can analyze the structure of each DEX instruction to show how it updates the taint information
We'll look at a few of the instruction and show how this is done
First, for the DEX move instruction, the semantics is like assignment in Java. The value of v_b is assigned to v_a
In terms of the taint update, this simply propagates the taint information from v_b to v_a
A unary operation works similarly: the actual semantics of the operator is ignored and the taint values are propegated
Finally, an array update stores some value v_a into array v_b at location v_c
Here we can see how the taint information of the entire array is stored into one value.
An array update takes the taint information of the entire array and then conjoins it with the taint information of the assigned variable
Operations for the remainder of the DEX language are defined in the paper
What's nice about this approach is we write simple rules for each instruction and by structural induction we get a way to define the taint flow for the entire program

Taint Propagation Logic

DEX Instruction	Semantics	Propagation
move v_a v_b	$v_a \leftarrow v_b$	$\tau(v_a) \leftarrow \tau(v_b)$
unary v_a v_b	$v_a \leftarrow \otimes v_a$	$\tau(v_a) \leftarrow \tau(v_b)$
aput v_a v_b v_c	$v_b[v_c] \leftarrow v_a$	$\tau(v_b[\cdot]) \leftarrow \tau(v_b[\cdot]) \cup \tau(v_a)$

Remaining operations are defined in the paper.

Given our new language, we can analyze the structure of each DEX instruction to show how it updates the taint information
We'll look at a few of the instruction and show how this is done
First, for the DEX move instruction, the semantics is like assignment in Java. The value of v_b is assigned to v_a
In terms of the taint update, this simply propagates the taint information from v_b to v_a
A unary operation works similarly: the actual semantics of the operator is ignored and the taint values are propegated
Finally, an array update stores some value v_a into array v_b at location v_c
Here we can see how the taint information of the entire array is stored into one value.
An array update takes the taint information of the entire array and then conjoins it with the taint information of the assigned variable
Operations for the remainder of the DEX language are defined in the paper
What's nice about this approach is we write simple rules for each instruction and by structural induction we get a way to define the taint flow for the entire program

Native Code Taint Propagation

- Native code taint propagation is unmonitored

We just defined propagation of taint tags inside the Dalvik VM
However, the taint propagation through native code is unmonitored
In order to maintain correctness, after the execution of some native function we need to update the taint tags of the return value and any class fields modified by the native call
There are two classes of native methods: internal VM methods, and those using the Java native interface.

Native Code Taint Propagation

- Native code taint propagation is unmonitored
- At native call, update taint tags of:

We just defined propagation of taint tags inside the Dalvik VM
However, the taint propagation through native code is unmonitored
In order to maintain correctness, after the execution of some native function we need to update the taint tags of the return value and any class fields modified by the native call
There are two classes of native methods: internal VM methods, and those using the Java native interface.

Native Code Taint Propagation

- Native code taint propagation is unmonitored
- At native call, update taint tags of:
  - Return value

We just defined propagation of taint tags inside the Dalvik VM
However, the taint propagation through native code is unmonitored
In order to maintain correctness, after the execution of some native function we need to update the taint tags of the return value and any class fields modified by the native call
There are two classes of native methods: internal VM methods, and those using the Java native interface.

Native Code Taint Propagation

- Native code taint propagation is unmonitored
- At native call, update taint tags of:
  - Return value
  - Any modified class fields

We just defined propagation of taint tags inside the Dalvik VM
However, the taint propagation through native code is unmonitored
In order to maintain correctness, after the execution of some native function we need to update the taint tags of the return value and any class fields modified by the native call
There are two classes of native methods: internal VM methods, and those using the Java native interface.

Native Code Taint Propagation

- Native code taint propagation is unmonitored
- At native call, update taint tags of:
  - Return value
  - Any modified class fields
- Two types of native methods:

We just defined propagation of taint tags inside the Dalvik VM
However, the taint propagation through native code is unmonitored
In order to maintain correctness, after the execution of some native function we need to update the taint tags of the return value and any class fields modified by the native call
There are two classes of native methods: internal VM methods, and those using the Java native interface.

Native Code Taint Propagation

- Native code taint propagation is unmonitored
- At native call, update taint tags of:
  - Return value
  - Any modified class fields
- Two types of native methods:
  - Internal VM methods

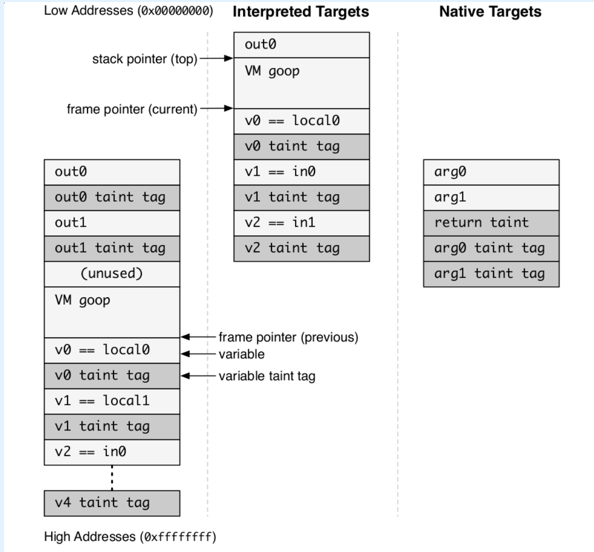
We just defined propagation of taint tags inside the Dalvik VM
However, the taint propagation through native code is unmonitored
In order to maintain correctness, after the execution of some native function we need to update the taint tags of the return value and any class fields modified by the native call
There are two classes of native methods: internal VM methods, and those using the Java native interface.

Native Code Taint Propagation

- Native code taint propagation is unmonitored
- At native call, update taint tags of:
  - Return value
  - Any modified class fields
- Two types of native methods:
  - Internal VM methods
  - JNI Methods

We just defined propagation of taint tags inside the Dalvik VM
However, the taint propagation through native code is unmonitored
In order to maintain correctness, after the execution of some native function we need to update the taint tags of the return value and any class fields modified by the native call
There are two classes of native methods: internal VM methods, and those using the Java native interface.

Internal VM Native Code Taint Propagation



An internal VM native call simply puts all the arguments passed to the function in an array of 32-bit registers

The middle of this figure shows the stack augmentation for internal VM calls

Although the stack layouts of the internal VM native calls and the normal Dalvik calls are similar, the key difference is that the internal VM calls are not running on the VM.

As such, the interpreter rules defined before will not be used

Internal VM Native Code Taint Propagation

- Patch internal VM native code to update taint info

To handle this case, the authors simply patched the internal VM functions to correctly use and update the taint information
At the time of writing, there were 185 internal VM native methods
After inspection, only 5 needed to be patched
And, since these methods are infrequently modified the amount of effort of this approach is minimal

Internal VM Native Code Taint Propagation

- Patch internal VM native code to update taint info
- 185 internal VM methods

To handle this case, the authors simply patched the internal VM functions to correctly use and update the taint information
At the time of writing, there were 185 internal VM native methods
After inspection, only 5 needed to be patched
And, since these methods are infrequently modified the amount of effort of this approach is minimal

Internal VM Native Code Taint Propagation

- Patch internal VM native code to update taint info
- 185 internal VM methods
- Only 5 needed to be patched

To handle this case, the authors simply patched the internal VM functions to correctly use and update the taint information
At the time of writing, there were 185 internal VM native methods
After inspection, only 5 needed to be patched
And, since these methods are infrequently modified the amount of effort of this approach is minimal

Internal VM Native Code Taint Propagation

- Patch internal VM native code to update taint info
- 185 internal VM methods
- Only 5 needed to be patched
- Infrequently modified

To handle this case, the authors simply patched the internal VM functions to correctly use and update the taint information
At the time of writing, there were 185 internal VM native methods
After inspection, only 5 needed to be patched
And, since these methods are infrequently modified the amount of effort of this approach is minimal

JNI Native Code Taint Propagation

- Java Native Interface Call Bridge

Next, I'll discuss how taint propagation occurs through native code using the Java native interface
The Java native interface call bridge parses the Java level arguments to be passed to native code and updates the return value
The authors modified the java native interface call bridge to correctly update the taint information of the return and class fields
To do this, they consult a method profile for the native call
The method profile is a list of pairs indicating flows between variables
For example, a native method writing to a file could be summarized as the taint information from the value being written flowing into the file
The issue with this is that the method profiles need to be created manually
They leave the automatic creation of method profiles for future work

JNI Native Code Taint Propagation

- Java Native Interface Call Bridge
  - Parses Java arguments

Next, I'll discuss how taint propagation occurs through native code using the Java native interface
The Java native interface call bridge parses the Java level arguments to be passed to native code and updates the return value
The authors modified the java native interface call bridge to correctly update the taint information of the return and class fields
To do this, they consult a method profile for the native call
The method profile is a list of pairs indicating flows between variables
For example, a native method writing to a file could be summarized as the taint information from the value being written flowing into the file
The issue with this is that the method profiles need to be created manually
They leave the automatic creation of method profiles for future work

JNI Native Code Taint Propagation

- Java Native Interface Call Bridge
  - Parses Java arguments
  - Assigns return value

Next, I'll discuss how taint propagation occurs through native code using the Java native interface
The Java native interface call bridge parses the Java level arguments to be passed to native code and updates the return value
The authors modified the java native interface call bridge to correctly update the taint information of the return and class fields
To do this, they consult a method profile for the native call
The method profile is a list of pairs indicating flows between variables
For example, a native method writing to a file could be summarized as the taint information from the value being written flowing into the file
The issue with this is that the method profiles need to be created manually
They leave the automatic creation of method profiles for future work

JNI Native Code Taint Propagation

- Java Native Interface Call Bridge
  - Parses Java arguments
  - Assigns return value
- Method Profile: list of $\langle \text{from}, \text{to} \rangle$

Next, I'll discuss how taint propagation occurs through native code using the Java native interface
The Java native interface call bridge parses the Java level arguments to be passed to native code and updates the return value
The authors modified the java native interface call bridge to correctly update the taint information of the return and class fields
To do this, they consult a method profile for the native call
The method profile is a list of pairs indicating flows between variables
For example, a native method writing to a file could be summarized as the taint information from the value being written flowing into the file
The issue with this is that the method profiles need to be created manually
They leave the automatic creation of method profiles for future work

JNI Native Code Taint Propagation

- Java Native Interface Call Bridge
  - Parses Java arguments
  - Assigns return value
- Method Profile: list of $\langle \text{from}, \text{to} \rangle$
  - `writeFile(name, val)`

Next, I'll discuss how taint propagation occurs through native code using the Java native interface
The Java native interface call bridge parses the Java level arguments to be passed to native code and updates the return value
The authors modified the java native interface call bridge to correctly update the taint information of the return and class fields
To do this, they consult a method profile for the native call
The method profile is a list of pairs indicating flows between variables
For example, a native method writing to a file could be summarized as the taint information from the value being written flowing into the file
The issue with this is that the method profiles need to be created manually
They leave the automatic creation of method profiles for future work

JNI Native Code Taint Propagation

- Java Native Interface Call Bridge
  - Parses Java arguments
  - Assigns return value
- Method Profile: list of $\langle$from, to$\rangle$
  - `writeFile(name, val)`
  - $\langle$from, to$\rangle = \langle$`val`, `name`$\rangle$

Next, I'll discuss how taint propagation occurs through native code using the Java native interface
The Java native interface call bridge parses the Java level arguments to be passed to native code and updates the return value
The authors modified the java native interface call bridge to correctly update the taint information of the return and class fields
To do this, they consult a method profile for the native call
The method profile is a list of pairs indicating flows between variables
For example, a native method writing to a file could be summarized as the taint information from the value being written flowing into the file
The issue with this is that the method profiles need to be created manually
They leave the automatic creation of method profiles for future work

JNI Native Code Taint Propagation

- Java Native Interface Call Bridge
  - Parses Java arguments
  - Assigns return value
- Method Profile: list of ⟨from, to⟩
  - `writeFile(name, val)`
  - ⟨from, to⟩ = ⟨`val`, `name`⟩
- Issue: method profile creation is manual and time consuming

Next, I'll discuss how taint propagation occurs through native code using the Java native interface
The Java native interface call bridge parses the Java level arguments to be passed to native code and updates the return value
The authors modified the java native interface call bridge to correctly update the taint information of the return and class fields
To do this, they consult a method profile for the native call
The method profile is a list of pairs indicating flows between variables
For example, a native method writing to a file could be summarized as the taint information from the value being written flowing into the file
The issue with this is that the method profiles need to be created manually
They leave the automatic creation of method profiles for future work

IPC Taint Propagation

- Propagate taint information from `send()` to `receive()`

thebloggerspost.wordpress

Next, we show how taint tags are propagated through IPC communications
This amounts to connecting the taint information from the location of a send in one application to a receive in another
As we've seen previously, we will again see a trade off between accuracy and scalability
Similar to how arrays are handled, the taint information of a message is aggregated into a single value
This approach is nice because it allows the taint information to correctly propagate regardless as to how the sender and receiver read and write the data
For example, the sender could aggregate an array of characters which are parsed as a single string by the receiver
However, just like in the array case using a single taint tag in a message can cause false alarms

IPC Taint Propagation

- Propagate taint information from `send()` to `receive()`
- Tracked on message level

thebloggerspost.wordpress

Next, we show how taint tags are propagated through IPC communications
This amounts to connecting the taint information from the location of a send in one application to a receive in another
As we've seen previously, we will again see a trade off between accuracy and scalability
Similar to how arrays are handled, the taint information of a message is aggregated into a single value
This approach is nice because it allows the taint information to correctly propagate regardless as to how the sender and receiver read and write the data
For example, the sender could aggregate an array of characters which are parsed as a single string by the receiver
However, just like in the array case using a single taint tag in a message can cause false alarms

IPC Taint Propagation

- Propagate taint information from `send()` to `receive()`
- Tracked on message level
- Correctly propagates regardless of message use

thebloggerspost.wordpress

Next, we show how taint tags are propagated through IPC communications
This amounts to connecting the taint information from the location of a send in one application to a receive in another
As we've seen previously, we will again see a trade off between accuracy and scalability
Similar to how arrays are handled, the taint information of a message is aggregated into a single value
This approach is nice because it allows the taint information to correctly propagate regardless as to how the sender and receiver read and write the data
For example, the sender could aggregate an array of characters which are parsed as a single string by the receiver
However, just like in the array case using a single taint tag in a message can cause false alarms

IPC Taint Propagation

- Propagate taint information from `send()` to `receive()`
- Tracked on message level
- Correctly propagates regardless of message use
  - Parse an array of characters as a single string

thebloggerspost.wordpress

Next, we show how taint tags are propagated through IPC communications
This amounts to connecting the taint information from the location of a send in one application to a receive in another
As we've seen previously, we will again see a trade off between accuracy and scalability
Similar to how arrays are handled, the taint information of a message is aggregated into a single value
This approach is nice because it allows the taint information to correctly propagate regardless as to how the sender and receiver read and write the data
For example, the sender could aggregate an array of characters which are parsed as a single string by the receiver
However, just like in the array case using a single taint tag in a message can cause false alarms

IPC Taint Propagation

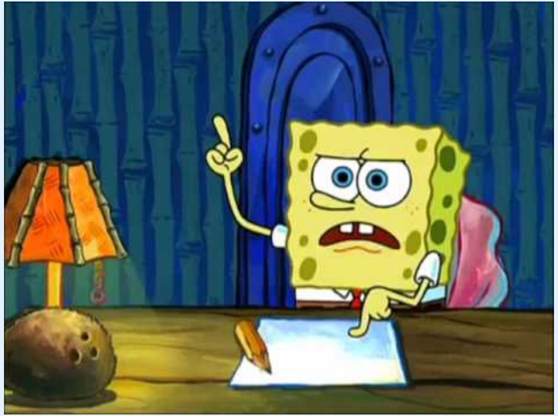
- Propagate taint information from `send()` to `receive()`
- Tracked on message level
- Correctly propagates regardless of message use
  - Parse an array of characters as a single string
- Aggregation leads to false alarms

thebloggerspost.wordpress

Next, we show how taint tags are propagated through IPC communications
This amounts to connecting the taint information from the location of a send in one application to a receive in another
As we've seen previously, we will again see a trade off between accuracy and scalability
Similar to how arrays are handled, the taint information of a message is aggregated into a single value
This approach is nice because it allows the taint information to correctly propagate regardless as to how the sender and receiver read and write the data
For example, the sender could aggregate an array of characters which are parsed as a single string by the receiver
However, just like in the array case using a single taint tag in a message can cause false alarms

File Taint Propegation

- Single tag per file
- Tainted write taints entire file



Finally, taint droid also propagates taint information through file reads and writes.
By now, this probably sounds old hat but we'll go over it anyway
Each file has a single taint tag
Similar to arrays, a tainted write to a file makes the entire file tainted
As a result, subsequent reads of the file will be tainted even if they do not actually read the tainted data.

Overview

Next, I'll go over their experimental evaluation

Experiments

- Analyzed 30 of most popular (2010) applications
- Needed to have suitable permissions
  - Access private data (source)
  - Access the internet (sink)
- Interesting results!

All in all, they analyzed 30 applications using TaintDroid
The applications were pulled from a list of the top 1,000 Android Market applications in 2010
They required the application have permissions suitable for analysis.
In other words, the application must have had permission to access some sensitive data, like the GPS, and also have access to the internet
Network connection was required since the only sinks considered were network sockets
The results found by the study, overall, were quite interesting
Two thirds of the applications have seemingly innocuous permissions requested at install but lead to private data leaks

Experiments

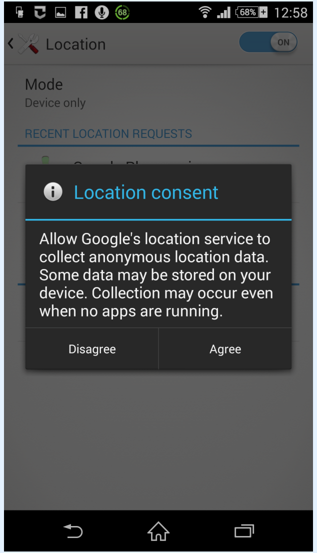
- Analyzed 30 of most popular (2010) applications
- Needed to have suitable permissions
  - Access private data (source)
  - Access the internet (sink)
- Interesting results!

Innocuous permissions can expose private data

All in all, they analyzed 30 applications using TaintDroid
The applications were pulled from a list of the top 1,000 Android Market applications in 2010
They required the application have permissions suitable for analysis.
In other words, the application must have had permission to access some sensitive data, like the GPS, and also have access to the internet
Network connection was required since the only sinks considered were network sockets
The results found by the study, overall, were quite interesting
Two thirds of the applications have seemingly innocuous permissions requested at install but lead to private data leaks

Data Monitoring

- Taint Droid running
- Application was installed and manually exercised
- Additional monitoring: IPC messages, network traffic
- Noted if application asked for users consent



To conduct the experiments, they installed and used the applications while taint droid was running
To provide an analysis of the results, the authors also monitored the contents of network and IPC traffic
They also noted if the application ever asked for user conset to send private data
An example of an application consenting to use private data is as in this image
The application is asking the user if it is OK to use their location

Data Monitoring

Applications*	#	Permissions†			
		L	C	A	P
The Weather Channel (News & Weather); Cestos, Solitaire (Game); Movies (Entertainment); Babble (Social); Manga Browser (Comics)	6	x			
Bump, Wertago (Social); Antivirus (Communication); ABC — Animals, Traffic Jam, Hearts, Blackjack, (Games); Horoscope (Lifestyle); Yellow Pages (Reference); 3001 Wisdom Quotes Lite, Dastelefonbuch, Astrid (Productivity), BBC News Live Stream (News & Weather); Ringtones (Entertainment)	14	x			x
Layar (Lifestyle); Knocking (Social); Coupons (Shopping); Trapster (Travel); Spongebob Slide (Game); ProBasketBall (Sports)	6	x	x		x
MySpace (Social); Barcode Scanner, ixMAT (Shopping)	3		x		
Evernote (Productivity)	1	x	x	x	

Here, we can see a table containing all the applications used
Many of them are fairly widely known such as MySpace, The Weather Channel, BBC News, and SpongeBob Slide
The four types of permissions are Location, Camera, Audio, and Phone state
Phone state includes access to stuff like SIM card ID, phone number, and other identifiers

Findings

Next, we'll discuss the findings of analyzing these 30 applications
Out of the twenty applications accessing phone state, many sent this information to external servers
They did not prompt the user to send such data
This shows the coarse grained permissions in android are not sufficient
Two of the twenty applications sent out IMSI, a unique identifier for a mobile subscriber, along with the geolocation of the user
The author theorizied IMSI was used as an identifier to build information about a users
A phone's IMEI uniquely identifiers the phone hardware.
9 applications sent out the IMEI over the network
7 of the 9 applications did not note IMEI harvesting in the license agreement
15 of the 30 applications sent loation data to advertisers; only 2 out of the 15 noted this in the EULA
One of the advantages of taint droid is presented in this analysis.
Since taint droid works at the variable level, it can detect location data transmitted both in binary and plaintext
This is more acurrate than a simple network monitoring approach
Finally, of the 105 alarms generated by TaintDroid, 39 were deemed benign after investigation

Findings

- 2 applications sent out IMSI and geolocation

Next, we'll discuss the findings of analyzing these 30 applications
Out of the twenty applications accessing phone state, many sent this information to external servers
They did not prompt the user to send such data
This shows the coarse grained permissions in android are not sufficient
Two of the twenty applications sent out IMSI, a unique identifier for a mobile subscriber, along with the geolocation of the user
The author theorizied IMSI was used as an identifier to build information about a users
A phone's IMEI uniquely identifiers the phone hardware.
9 applications sent out the IMEI over the network
7 of the 9 applications did not note IMEI harvesting in the license agreement
15 of the 30 applications sent loation data to advertisers; only 2 out of the 15 noted this in the EULA
One of the advantages of taint droid is presented in this analysis.
Since taint droid works at the variable level, it can detect location data transmitted both in binary and plaintext
This is more acurrate than a simple network monitoring approach
Finally, of the 105 alarms generated by TaintDroid, 39 were deemed benign after investigation

Findings

- 2 applications sent out IMSI and geolocation
- 9 applications sent out IMEI

Next, we'll discuss the findings of analyzing these 30 applications
Out of the twenty applications accessing phone state, many sent this information to external servers
They did not prompt the user to send such data
This shows the coarse grained permissions in android are not sufficient
Two of the twenty applications sent out IMSI, a unique identifier for a mobile subscriber, along with the geolocation of the user
The author theorizied IMSI was used as an identifier to build information about a users
A phone's IMEI uniquely identifiers the phone hardware.
9 applications sent out the IMEI over the network
7 of the 9 applications did not note IMEI harvesting in the license agreement
15 of the 30 applications sent loation data to advertisers; only 2 out of the 15 noted this in the EULA
One of the advantages of taint droid is presented in this analysis.
Since taint droid works at the variable level, it can detect location data transmitted both in binary and plaintext
This is more acurrate than a simple network monitoring approach
Finally, of the 105 alarms generated by TaintDroid, 39 were deemed benign after investigation

Findings

- 2 applications sent out IMSI and geolocation
- 9 applications sent out IMEI
- 15 applications send location information to advertisers

Next, we'll discuss the findings of analyzing these 30 applications
Out of the twenty applications accessing phone state, many sent this information to external servers
They did not prompt the user to send such data
This shows the coarse grained permissions in android are not sufficient
Two of the twenty applications sent out IMSI, a unique identifier for a mobile subscriber, along with the geolocation of the user
The author theorizied IMSI was used as an identifier to build information about a users
A phone's IMEI uniquely identifiers the phone hardware.
9 applications sent out the IMEI over the network
7 of the 9 applications did not note IMEI harvesting in the license agreement
15 of the 30 applications sent loation data to advertisers; only 2 out of the 15 noted this in the EULA
One of the advantages of taint droid is presented in this analysis.
Since taint droid works at the variable level, it can detect location data transmitted both in binary and plaintext
This is more acurrate than a simple network monitoring approach
Finally, of the 105 alarms generated by TaintDroid, 39 were deemed benign after investigation

Findings

- 2 applications sent out IMSI and geolocation
- 9 applications sent out IMEI
- 15 applications send location information to advertisers
  - Sent over both binary or plaintext

Next, we'll discuss the findings of analyzing these 30 applications
Out of the twenty applications accessing phone state, many sent this information to external servers
They did not prompt the user to send such data
This shows the coarse grained permissions in android are not sufficient
Two of the twenty applications sent out IMSI, a unique identifier for a mobile subscriber, along with the geolocation of the user
The author theorizied IMSI was used as an identifier to build information about a users
A phone's IMEI uniquely identifiers the phone hardware.
9 applications sent out the IMEI over the network
7 of the 9 applications did not note IMEI harvesting in the license agreement
15 of the 30 applications sent loation data to advertisers; only 2 out of the 15 noted this in the EULA
One of the advantages of taint droid is presented in this analysis.
Since taint droid works at the variable level, it can detect location data transmitted both in binary and plaintext
This is more acurrate than a simple network monitoring approach
Finally, of the 105 alarms generated by TaintDroid, 39 were deemed benign after investigation

Findings

- 2 applications sent out IMSI and geolocation
- 9 applications sent out IMEI
- 15 applications send location information to advertisers
  - Sent over both binary or plaintext
- Of 105 flagged alarms, 39 were deemed to be benign

Next, we'll discuss the findings of analyzing these 30 applications
Out of the twenty applications accessing phone state, many sent this information to external servers
They did not prompt the user to send such data
This shows the coarse grained permissions in android are not sufficient
Two of the twenty applications sent out IMSI, a unique identifier for a mobile subscriber, along with the geolocation of the user
The author theorizied IMSI was used as an identifier to build information about a users
A phone's IMEI uniquely identifiers the phone hardware.
9 applications sent out the IMEI over the network
7 of the 9 applications did not note IMEI harvesting in the license agreement
15 of the 30 applications sent loation data to advertisers; only 2 out of the 15 noted this in the EULA
One of the advantages of taint droid is presented in this analysis.
Since taint droid works at the variable level, it can detect location data transmitted both in binary and plaintext
This is more acurrate than a simple network monitoring approach
Finally, of the 105 alarms generated by TaintDroid, 39 were deemed benign after investigation

Performance Evaluations

- Nexus One with Android 2.1

Next, we look at the performance costs of using TaintDroid
Overall, the overhead was fairly low
The authors believe this ocurred since most applications are just waiting for the user to do something, and that most of the complex code, such as screen rendering, is in native libraries
The authors created a few macro benchmarks performing some tasks like reading/writing to the address book, making a phone call, or taking pictures
The most overhead was 29% for the picture
The authors note that this ocurrs due to the overhead from additional file write operations for tainted data

Performance Evaluations

- Nexus One with Android 2.1
- Overhead was fairly low:

Next, we look at the performance costs of using TaintDroid
Overall, the overhead was fairly low
The authors believe this ocurred since most applications are just waiting for the user to do something, and that most of the complex code, such as screen rendering, is in native libraries
The authors created a few macro benchmarks performing some tasks like reading/writing to the address book, making a phone call, or taking pictures
The most overhead was 29% for the picture
The authors note that this ocurrs due to the overhead from additional file write operations for tainted data

Performance Evaluations

- Nexus One with Android 2.1
- Overhead was fairly low:
  - Most applications are just waiting for the user

Next, we look at the performance costs of using TaintDroid
Overall, the overhead was fairly low
The authors believe this ocurred since most applications are just waiting for the user to do something, and that most of the complex code, such as screen rendering, is in native libraries
The authors created a few macro benchmarks performing some tasks like reading/writing to the address book, making a phone call, or taking pictures
The most overhead was 29% for the picture
The authors note that this ocurrs due to the overhead from additional file write operations for tainted data

Performance Evaluations

- Nexus One with Android 2.1
- Overhead was fairly low:
  - Most applications are just waiting for the user
  - Complex code is in native libraries

Next, we look at the performance costs of using TaintDroid
Overall, the overhead was fairly low
The authors believe this ocurred since most applications are just waiting for the user to do something, and that most of the complex code, such as screen rendering, is in native libraries
The authors created a few macro benchmarks performing some tasks like reading/writing to the address book, making a phone call, or taking pictures
The most overhead was 29% for the picture
The authors note that this ocurrs due to the overhead from additional file write operations for tainted data

Performance Evaluations

- Nexus One with Android 2.1
- Overhead was fairly low:
  - Most applications are just waiting for the user
  - Complex code is in native libraries

Next, we look at the performance costs of using TaintDroid
Overall, the overhead was fairly low
The authors believe this ocurred since most applications are just waiting for the user to do something, and that most of the complex code, such as screen rendering, is in native libraries
The authors created a few macro benchmarks performing some tasks like reading/writing to the address book, making a phone call, or taking pictures
The most overhead was 29% for the picture
The authors note that this ocurrs due to the overhead from additional file write operations for tainted data

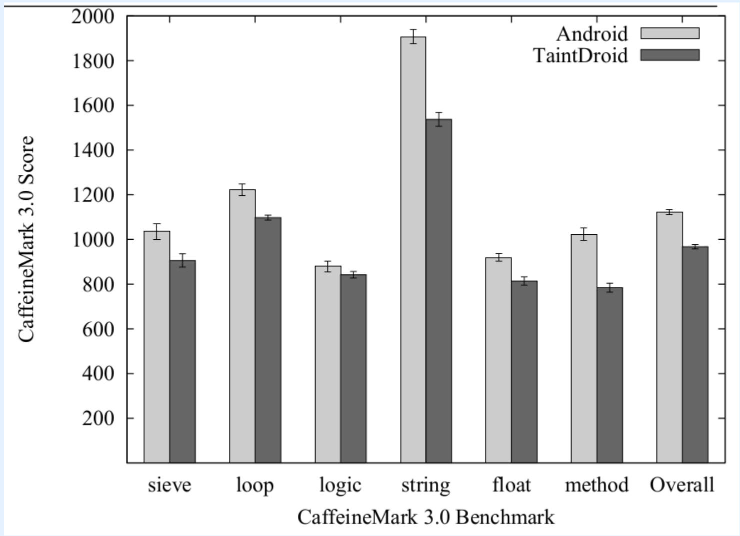
Performance Evaluations

- Nexus One with Android 2.1
- Overhead was fairly low:
  - Most applications are just waiting for the user
  - Complex code is in native libraries

	Android	TaintDroid
App Load Time	63 ms	65 ms
Address Book (create)	348 ms	367 ms
Address Book (read)	101 ms	119 ms
Phone Call	96 ms	106 ms
Take Picture	1718 ms	2216 ms

Next, we look at the performance costs of using TaintDroid
Overall, the overhead was fairly low
The authors believe this ocurred since most applications are just waiting for the user to do something, and that most of the complex code, such as screen rendering, is in native libraries
The authors created a few macro benchmarks performing some tasks like reading/writing to the address book, making a phone call, or taking pictures
The most overhead was 29% for the picture
The authors note that this ocurrs due to the overhead from additional file write operations for tainted data

Microbenchmark: CaffineMark3.0



Next, the authors tested taint droid on a Java microbenchmark called CaffeineMark
CaffineMark has its own relative scoring metric which rougly corresponds to the number of instructions per second
TaintDroid, as expected, as small overhead for those involving arithmetic
These cases are simple for taint droid since they only involve single spatially located taint tags for local variables
Overall, the overhead was about 14% for TaintDroid
The memory overhead as 4.4%

IPC Overhead

	Android	TaintDroid
Time (s)	8.58	10.89
Memory (client)	21.06MB	21.88MB
Memory (service)	18.92MB	19.48MB

Next, the authors tested the IPC overhead of TaintDroid
To do this, they created a client–service microbenchmark where the client requests the service to update some data
They repeated this many many times and checked the overhead
Overall, both the time and space overheads are pretty small
The test was 27% slower and used 3.5% more memory

Conclusion

- Efficient, system-wide, dynamic taint tracking

In conclusion, the authors presented taint droid, an efficient system wide dynamic taint tracking implementation
Their method allows for multiple taint sources to be tracked simultaneously
Experimental results show that it has a low time and space overhead
And, by analyzing popular applications they showed that android permissions may be too coarse to provide adequet privacy.

Conclusion

- Efficient, system-wide, dynamic taint tracking
- Simultaneously track multiple taint sources

In conclusion, the authors presented taint droid, an efficient system wide dynamic taint tracking implementation
Their method allows for multiple taint sources to be tracked simultaneously
Experimental results show that it has a low time and space overhead
And, by analyzing popular applications they showed that android permissions may be too coarse to provide adequet privacy.

Conclusion

- Efficient, system-wide, dynamic taint tracking
- Simultaneously track multiple taint sources
- Low time and space overhead

In conclusion, the authors presented taint droid, an efficient system wide dynamic taint tracking implementation
Their method allows for multiple taint sources to be tracked simultaneously
Experimental results show that it has a low time and space overhead
And, by analyzing popular applications they showed that android permissions may be too coarse to provide adequet privacy.

Conclusion

- Efficient, system-wide, dynamic taint tracking
- Simultaneously track multiple taint sources
- Low time and space overhead
- Android permissions are too coarse

In conclusion, the authors presented taint droid, an efficient system wide dynamic taint tracking implementation
Their method allows for multiple taint sources to be tracked simultaneously
Experimental results show that it has a low time and space overhead
And, by analyzing popular applications they showed that android permissions may be too coarse to provide adequet privacy.

Conclusion

- Efficient, system-wide, dynamic taint tracking
- Simultaneously track multiple taint sources
- Low time and space overhead
- Android permissions are too coarse

In conclusion, the authors presented taint droid, an efficient system wide dynamic taint tracking implementation
Their method allows for multiple taint sources to be tracked simultaneously
Experimental results show that it has a low time and space overhead
And, by analyzing popular applications they showed that android permissions may be too coarse to provide adequet privacy.

Conclusion

- Efficient, system-wide, dynamic taint tracking
- Simultaneously track multiple taint sources
- Low time and space overhead
- Android permissions are too coarse

Questions?

In conclusion, the authors presented taint droid, an efficient system wide dynamic taint tracking implementation
Their method allows for multiple taint sources to be tracked simultaneously
Experimental results show that it has a low time and space overhead
And, by analyzing popular applications they showed that android permissions may be too coarse to provide adequet privacy.

Discussion

- Does not track implicit flows

Finally, we discuss some of the good and bad of taint droid
First, TaintDroid does not follow implicit flows, i.e., those through control
We had an example of this earlier in the semester where an information leak could ocurr through open or closing the CD drive
Next, taint droid does not track taint information traveling through the network.
In this way, the attacker could send tainted data over the network and Taint Droid would miss the taint propegation from a network read of the same value
Finally, the authors note that when taint information is stored with un-tainted, commonly used values many false alarms can be generted
For example, a single string contains a sensitive value along with other non-sensitive data

Discussion

- Does not track implicit flows
- Does not track taint information returning through the network

Finally, we discuss some of the good and bad of taint droid
First, TaintDroid does not follow implicit flows, i.e., those through control
We had an example of this earlier in the semester where an information leak could ocurr through open or closing the CD drive
Next, taint droid does not track taint information traveling through the network.
In this way, the attacker could send tainted data over the network and Taint Droid would miss the taint propegation from a network read of the same value
Finally, the authors note that when taint information is stored with un-tainted, commonly used values many false alarms can be generted
For example, a single string contains a sensitive value along with other non-sensitive data

Discussion

- Does not track implicit flows
- Does not track taint information returning through the network
- Taint-tracking granularities lead to false positive

Finally, we discuss some of the good and bad of taint droid
First, TaintDroid does not follow implicit flows, i.e., those through control
We had an example of this earlier in the semester where an information leak could ocurr through open or closing the CD drive
Next, taint droid does not track taint information traveling through the network.
In this way, the attacker could send tainted data over the network and Taint Droid would miss the taint propegation from a network read of the same value
Finally, the authors note that when taint information is stored with un-tainted, commonly used values many false alarms can be generted
For example, a single string contains a sensitive value along with other non-sensitive data

Discussion

- Does not track implicit flows
- Does not track taint information returning through the network
- Taint-tracking granularities lead to false positive
  - E.g., Tainted information stored with non-tainted information

Finally, we discuss some of the good and bad of taint droid
First, TaintDroid does not follow implicit flows, i.e., those through control
We had an example of this earlier in the semester where an information leak could ocurr through open or closing the CD drive
Next, taint droid does not track taint information traveling through the network.
In this way, the attacker could send tainted data over the network and Taint Droid would miss the taint propegation from a network read of the same value
Finally, the authors note that when taint information is stored with un-tainted, commonly used values many false alarms can be generted
For example, a single string contains a sensitive value along with other non-sensitive data

Security Concerns

- Trusted Code Base:
  - Virtual Machine
  - Native Code Libraries (`.so` files)
- Only way to escape VM is through native code
- Prevent third-party applications from using their own `.so` files

Sleeping While on Duty (wikipedia)

Next, I'll discuss their security assumptions
They assume the virtual machine and native code libraries are trusted
The third-party applications are confined to the VM for most of their operation.
As a result, the third-party application cannot attack their taint tracker in Java mode
But, through Native code they could potentially do malicious things to the running VM
To handle this, they prevent third-party libraries from executing non-system native code

Taint Interface Library

- `addTaint()`

Finally, we get to how the taint tracking library can be used
The developer passes a variable to a certain addTaint function which updates the taint tag associated with the variable
This value is then propagated using the rules we previously described
They do not allow arbitrary sets of taint values since the function is called in an untrusted environment
In other words, you can only taint a value in the untrusted Java environment and not un-taint it
Taint sources are identified by the user and then automatically instrumented
Some taint sources include SMS databases, sensors, and device identifiers like phone number, or SIM card ID
For sinks, they used network sockets

Taint Interface Library

- `addTaint()`
- Can only add taint markings to variables

Finally, we get to how the taint tracking library can be used
The developer passes a variable to a certain addTaint function which updates the taint tag associated with the variable
This value is then propagated using the rules we previously described
They do not allow arbitrary sets of taint values since the function is called in an untrusted environment
In other words, you can only taint a value in the untrusted Java environment and not un-taint it
Taint sources are identified by the user and then automatically instrumented
Some taint sources include SMS databases, sensors, and device identifiers like phone number, or SIM card ID
For sinks, they used network sockets

Taint Interface Library

- `addTaint()`
- Can only add taint markings to variables
- No arbitrary sets

Finally, we get to how the taint tracking library can be used
The developer passes a variable to a certain addTaint function which updates the taint tag associated with the variable
This value is then propagated using the rules we previously described
They do not allow arbitrary sets of taint values since the function is called in an untrusted environment
In other words, you can only taint a value in the untrusted Java environment and not un-taint it
Taint sources are identified by the user and then automatically instrumented
Some taint sources include SMS databases, sensors, and device identifiers like phone number, or SIM card ID
For sinks, they used network sockets

Taint Interface Library

- `addTaint()`
- Can only add taint markings to variables
- No arbitrary sets
- Taint sources are identified by user and automatically instrumented

Finally, we get to how the taint tracking library can be used
The developer passes a variable to a certain addTaint function which updates the taint tag associated with the variable
This value is then propagated using the rules we previously described
They do not allow arbitrary sets of taint values since the function is called in an untrusted environment
In other words, you can only taint a value in the untrusted Java environment and not un-taint it
Taint sources are identified by the user and then automatically instrumented
Some taint sources include SMS databases, sensors, and device identifiers like phone number, or SIM card ID
For sinks, they used network sockets

Taint Interface Library

- `addTaint()`
- Can only add taint markings to variables
- No arbitrary sets
- Taint sources are identified by user and automatically instrumented
  - SMS databases

Finally, we get to how the taint tracking library can be used
The developer passes a variable to a certain addTaint function which updates the taint tag associated with the variable
This value is then propagated using the rules we previously described
They do not allow arbitrary sets of taint values since the function is called in an untrusted environment
In other words, you can only taint a value in the untrusted Java environment and not un-taint it
Taint sources are identified by the user and then automatically instrumented
Some taint sources include SMS databases, sensors, and device identifiers like phone number, or SIM card ID
For sinks, they used network sockets

Taint Interface Library

- `addTaint()`
- Can only add taint markings to variables
- No arbitrary sets
- Taint sources are identified by user and automatically instrumented
  - SMS databases
  - Sensors (microphone, GPS)

Finally, we get to how the taint tracking library can be used
The developer passes a variable to a certain addTaint function which updates the taint tag associated with the variable
This value is then propagated using the rules we previously described
They do not allow arbitrary sets of taint values since the function is called in an untrusted environment
In other words, you can only taint a value in the untrusted Java environment and not un-taint it
Taint sources are identified by the user and then automatically instrumented
Some taint sources include SMS databases, sensors, and device identifiers like phone number, or SIM card ID
For sinks, they used network sockets

Taint Interface Library

- `addTaint()`
- Can only add taint markings to variables
- No arbitrary sets
- Taint sources are identified by user and automatically instrumented
  - SMS databases
  - Sensors (microphone, GPS)
  - Device identifiers

Finally, we get to how the taint tracking library can be used
The developer passes a variable to a certain addTaint function which updates the taint tag associated with the variable
This value is then propagated using the rules we previously described
They do not allow arbitrary sets of taint values since the function is called in an untrusted environment
In other words, you can only taint a value in the untrusted Java environment and not un-taint it
Taint sources are identified by the user and then automatically instrumented
Some taint sources include SMS databases, sensors, and device identifiers like phone number, or SIM card ID
For sinks, they used network sockets

Taint Interface Library

- `addTaint()`
- Can only add taint markings to variables
- No arbitrary sets
- Taint sources are identified by user and automatically instrumented
  - SMS databases
  - Sensors (microphone, GPS)
  - Device identifiers
- Sinks: network socket

Finally, we get to how the taint tracking library can be used
The developer passes a variable to a certain addTaint function which updates the taint tag associated with the variable
This value is then propagated using the rules we previously described
They do not allow arbitrary sets of taint values since the function is called in an untrusted environment
In other words, you can only taint a value in the untrusted Java environment and not un-taint it
Taint sources are identified by the user and then automatically instrumented
Some taint sources include SMS databases, sensors, and device identifiers like phone number, or SIM card ID
For sinks, they used network sockets