"TrustDroid™": Preventing the use of SmartPhones for information leaking in corporate networks through the used of static analysis taint tracking

Zhibo Zhao and Fernando C. Colon Osorio
Wireless Systems Security Research Laboratory and Brandeis University
zhaozb@brandeis.edu, fcco@brandeis.edu

Abstract

Over the last 12 years three important dates have marked the beginning of a major paradigm shift in computing and the security models applied to protect an emerging computing environment - March 1999, January 9th, 2007, and July 2007. These dates roughly correspond to the birth of SalesForce.com, the most successful Software as a Service (SaS) provider to date, Steve Jobs introduction of the Iphone,, and the discovery of the Zeus Botnet. These innovations have been instrumental in enabling a paradigm shift in computing, away from a corporate network centric model with Windows end-point devices to what we called in this manuscript the Circa 2020 Computing Model. In the circa 2020 Computing model applications and data reside in the Cloud, the concept of an extended Trust Domain (network) disappears - there is no corporate network, and finally the end-point device is a SmartPhone owned and operated by employees - Bring Your Own Device (BYOD). In such an environment, the end-point device is not "Trusted", and there is a high likelihood that the BYOD can be used as a channel to leak sensitive data. In this manuscript, we present a new mechanism to prevent such a situation. We called this mechanism "TrustDroid™". TrustDroid™ is a static analyzer based on taint tracking that can be used to prevent leakage of sensitive information by an un-trusted Android SmartPhone.

1. Introduction

Over the last 12 years three important dates have marked the beginnings of a major paradigm shift in computing and the security models applied to protect an emerging computing environment - March 1999, January 9th, 2007, and July 2007.These dates roughly correspond to the birth of SalesForce.com, the most successful Software as a Service (SaS) provider to date, Steve Jobs introduction of the Iphone,, and the discovery of the Zeus Botnet. These innovations have been instrumental in enabling a paradigm shift in computing, away from a corporate network centric model with Windows end-point devices to what we called in this manuscript the Circa 2020 Computing Model. In the circa 202 Computing model applications and data reside in the Cloud, the concept of an extended Trust Domain (network) disappears - there are no barriers to protect when your data and applications reside with your vendors, and the end-point device is a SmartPhone owned and operated by your employees- Bring Your Own Device (BYOD).

In this paper we describe TrustDroid™, a static analyzer based on taint tracking, as a solution to the security concerns created by the paradigm shift described above. In our model, we assume that the Android device (BYOD) may run un-trusted applications while accessing corporate private information, and therefore sensitive information may be leaked. TrustDroid™ statically performs semantic analysis of a compiled Android application (APK file). Having perform such static analysis, it will then determine if leakage of sensitive information is possible. If such a possibility exists a warning of information leakage is delivered to the user of the device.

TrustDroid™ can operate in two modes - off-line and real time. In offline mode, corporate resources are brought to bear to the static analysis problem, and hence performance is not a problem. However, if we want TrustDroid™ to operate real-time, then, performance of the algorithms, both in term of speed and battery/resource

978-1-4673-4879-9/12/$31.00 ©2012 IEEE

consumption become paramount. Unfortunately, see [14], traditional taint-checking methodology incurs unacceptable CPU and memory utilization overheads that will make such approach infeasible. Instead TrustDroid™ takes advantage of the Dalvik virtual machine present in the Android environment to significantly reduce this taint tracking overhead. In section 4, we described how this is accomplished. This manuscript makes the following contributions. First, a solution to corporate sensitive information leakage (ex-filtration), called TrustDroid™, is presented. TrustDroid™ works in standalone mode, and it requires neither OS support nor source code modifications, and it is therefore easier to implement for real-world usage than previously known solutions.

The reminder of this manuscript is organized as follows. Section 2 presents a brief introduction of the Android platform, and provides the necessary background to the reader in order to understand the technical underpinnings of TrustDroid™. Section 3 presents an overview of the TrustDroid™ approach. Section 4 presents a detailed design and implementation of the system. Section 5 demonstrates how TrustDroid™ works, and finally Section 6 describes future work.

2. Overview of the Android environment

Android is a Linux-based operating system for mobile devices. Android applications are developed for and run on a Java Virtual Machine customized for devices with limited memory and CPU speed. The VM is named Dalvik, Android applications are converted from Java Virtual Machine-compatible .class files to Dalvik-compatible .dex (Dalvik Executable) files before installation on an Android device.

Unlike most Java VMs, which are based on a stack architecture, the Dalvik VM is a register-to-register ISP, and therefore all computation is performed solely on registers. Values must be loaded from and stored to class fields before use and after use. Dalvik uses class fields for all long term storage, unlike hardware register-based machine which store values in arbitrary memory locations. Every method of the Dalvik VM exclusively owns a set of registers. Registers are used to store local variables and argument variables. Dalvik VM uses two naming schemes for registers - the normal v-naming scheme and the p-naming scheme for parameter registers. The first register in the p-naming scheme is the first parameter register in the method. So if a method has 2 arguments and 4 total registers, then, the naming schema is shown in the following table.

v0		first local register
v1		second local register
v2	p0	first parameter register
v3	p1	second parameter register

In addition to Dalvik Virtual Machine, Android also provides access to native libraries for performance optimization and third-party libraries. Android Native code is written in C/C++ ,supported and exposed to Dalvik by the Linux kernel and its services. Dalvik uses JNI interface to call native code, while at the same time native code can make calls back to the Dalvik domain. Generally speaking there are two kinds of native methods. These are: (1) system native library methods, and (2) the user developed native library.

3. Methodology and Approach

We are seeking a system-wide approach to monitor the data flow of sensitive information while delivering reasonable performance. The approach is to basically use static semantic analysis on compiled Android byte code to perform data flow tracking. TrustDroid™ analyzes the byte code in search of entries that manipulate sensitive data as specified in the sensitive information source set (see more details in section 5). Any such sensitive data found will be initially marked as tainted with a taint tag, and this tag propagates when the data is manipulated by the Bytecode, such as copying one variable to another variable, or to another memory location through a function call. Finally if tainted data flows out through a pre-defined taint sinks (such

as network interface), then, such operation will be flagged as a potential leak activity by the Bytecode.

The basic mechanism of taint checking is tracking the flow of data transportation through every instruction executed. However, in order for such an approach to be feasible two basic challenges must be overcome. The first challenge is the classic problem of static versus run-time analysis, meaning you need to correctly predict the instruction sequence execution from the compiled APK file. Thanks to the well-formed DEX file format and the Dalvik instruction set, we have been able to design and built an engine which is able to restore DEX file to a text format that contains not only instruction sequences but also the directives of the program. We have challenged the correctness of our approach by running a large number of tests, and have been able to verified that the text format produced by our engine can always be compiled back to Dex format. The second challenge is to correctly understand the semantics of the instructions. The situation is harder for x86 instructions set, but much better if the Dalvik instructions set is used. The Dalvik instruction set itself contains its semantics [4]. Hence, strictly-formed Dalvik instructions makes the job of extracting the semantics a lot easier.

The final challenge to our approach is the run time performance of taint checking. Traditional taint checking causes severe performance degradation. We alleviate this problem in the following ways. First, our taint tracking is implemented at different selectable levels of granularity. For example, in certain scenarios, we will choose a relatively coarse granular tracking level to purse a balance between false-positive and performance. Secondly, TrustDroid™ produces a well-formed intermediate file and therefore accurately tracks the data flow without the overhead of running it in a sandbox. Lastly, thanks to Android's well-defined interface, sensitive data sources and taint sinks are abstracted to a simple high level interface, which to a great extend reduces the workload associated with traditional taint checking systems.

A word of caution. Currently, our designs works only with the Dalvik interface. Hence, if other programming interfaces present in the Android platform are used, such as user-defined JNI, then our work is no longer applicable. Having said this and in theory, the approach taken here can be replicated to accommodate such cases.. Further, and according to real-world statistics, see Enck et. al. [1], the number of applications that use non-Dalvik interface is rather small, hence, our approach can be practically applied to the majority of existing real life cases.

4. TrustDroid™

In this section we will introduce the detailed design and implementation of the TrustDroid™ system from the aspects of semantic analyzing, taint propagation logic, and taint storage management.

4.1 The Semantic Analyzer

Semantic analysis begins by processing raw Dex file. Based on the file format of dex file [2], we build a parser based on the open-sourced parser generator ANTLR[3], to parse the structure of a given Dex file. The token set used by the parser is build based on the Dalvik byte codes [4]. As a result, a given Dex file compiled from source code as in Figure 1 will generate a tree structure shown here in Figure 2.

Note that the sample source code in Figure 1 retrieves local phone number (sensitive data) and sends it out through a wireless network. We will use this code as our example for the remaining of this section.

Apparently in figure 2, by manually adding 'virtual nodes' (nodes with name starting with I_ are virtual nodes that represent a structure of source code but not corresponding to anything in original Dex file), the tree structure contains not only the executable byte code stream but also the structure of Java source code.

Therefore we continuously build another tree parser, which converts the tree structure in Figure 2 into text based descriptions as in Figure 3. The

text is basically in Jasmin syntax format [5] and we borrow tokens and directives from Smali[6].

```
/*retrieve local phone number*/
            TelephonyManager phoneMgr=(TelephonyManager)
this.getSystemService(Context.TELEPHONY_SERVICE);
            String number=phoneMgr.getLine1Number();

            Trustdroid_test.sendinfo(number);


class Trustdroid_test
{
    public static void sendinfo(String phoneNumber) throws
UnknownHostException, IOException
    {
        Socket socket=new Socket("127.0.0.1",123);
        OutputStreamWriter writer=new OutputStreamWriter
(socket.getOutputStream());
        writer.write(phoneNumber);
    }
}
```

Figure 1: source code segment of a sample program

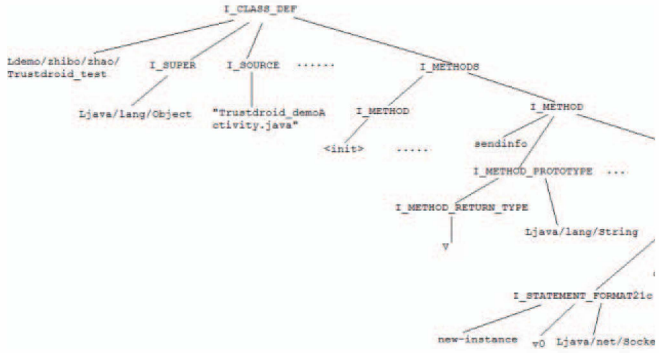

Figure 2: tree structure generated by source code of class TrustDroid_test

The value of the text shown above is that such text will serve as the basis for static taint tracking. The text exhibits the following properties: (1) everything is strong typed, making taint tracking easy (2) instructions strictly match Dalvik opcode (3) it is so close to source code that it embeds the program call graph in it. Note that all branch instructions have a label rather than a variable as operand, making the text itself a good representation of the call graph of the program. This call graph forms the basis for our taint tracking system.

4.2 Taint Propagation Rule

4.2.1 Primitive data taint propagation rule

Syntactically there are three types of operands in the Dalvik instruction set: register (V), immediate 16, 32 bit, and double-width (64-bit) constants (#), and type index (index@type). Semantically however they can represent four distinct types of data: local variable as V, argument variable as V, class field as v->index@type, and static field(index@type). See more details at [4].

```
.class Ldemo/zhibo/zhao/Trustdroid_test;
.super Ljava/lang/Object;
.source "Trustdroid_demoActivity.java"


# direct methods
.method public static sendinfo(Ljava/lang/String;)V
    .registers 5
    .parameter "phoneNumber"
    .annotation system Ldalvik/annotation/Throws;
        value = {
            Ljava/net/UnknownHostException;,
            Ljava/io/IOException;
        }
    .end annotation

    .prologue
    .line 42
    new-instance v0, Ljava/net/Socket;

    const-string v2, "127.0.0.1"

    const/16 v3, 0x7b

    invoke-direct {v0, v2, v3}, Ljava/net/Socket;-><init>
(Ljava/lang/String;I)V

    .line 43
    .local v0, socket:Ljava/net/Socket;
    new-instance v1, Ljava/io/OutputStreamWriter;

    invoke-virtual {v0}, Ljava/net/Socket;->getOutputStream()
Ljava/io/OutputStream;

    move-result-object v2

    invoke-direct {v1, v2}, Ljava/io/OutputStreamWriter;-><init>
(Ljava/io/OutputStream;)V

    .line 44
    .local v1, writer:Ljava/io/OutputStreamWriter;
    invoke-virtual {v1, p0}, Ljava/io/OutputStreamWriter;->write
(Ljava/lang/String;)V

    .line 45
    return-void
.end method
```

Figure 3.1 Jasmin syntax of class TrustDroid_test

```
    const-string v3, "phone"

    invoke-virtual {p0, v3},
Ldemo/zhibo/zhao/Trustdroid_demoActivity;->getSystemService
(Ljava/lang/String;)Ljava/lang/Object;

    move-result-object v2

    check-cast v2, Landroid/telephony/TelephonyManager;

    .line 24
    .local v2, phoneMgr:Landroid/telephony/TelephonyManager;
    invoke-virtual {v2}, Landroid/telephony/TelephonyManager;->
getLine1Number()Ljava/lang/String;

    move-result-object v1

    .line 26
    .local v1, number:Ljava/lang/String;
    invoke-static {v1}, Ldemo/zhibo/zhao/Trustdroid_test;->
sendinfo(Ljava/lang/String;)V
```

Figure 3.2 Jasmin syntax of source code segment depicted in Figure 1

As we said before, Dalvik instructions are so well semantically categorized that it is rather straightforward to define the taint propagation rules for instructions that move data across the

four five types described above. For example, consider the Dalvik instruction below:

iput vA, vB, field@CCCC

The semantics of this instruction specify that the value of instance field identified by field@CCCC of the object identified by vB should be moved to register vA. In this case the taint propagation rule will read: if the value of vB->field@CCCC is tainted, then taint vA after the input operation.

The propagation rule for method invoke instructions is also straightforward. Take one instruction that invokes a virtual methods in Figure 3 as example:

```
    invoke-virtual {p0, v3},
Ldemo/zhibo/zhao/Trustdroid_demoActivity;->getSystemService
(Ljava/lang/String;)Ljava/lang/Object;

    move-result-object v2
```

3: Virtual Method Invocation and Tainting

Semantically, the index of called method in this instance is present as

Ldemo/Zhibo/zhao/TrustDroid_demoActivity;->getSystemService,

meaning the instruction is calling

getSystemSerive of an instance of type *demo/Zhibo/zhao/Trustdroid_demoActivity*.

The beginning L represents the starting of a class type and the semicolon its end. The reference of the object instance is stored in p0. Note p0 is actually an alias of vN while N depends on the context. The argument of the call is stored in v3. The method prototype
(Ljava/lang/String;)Ljava/lang/Object;
implies that there is only one argument of String type. The move-result-object instruction is an exception as it is always combined with the previous instruction. (See more detail of Jasmine syntax in [5][6]).

Based on the semantics above, we can assert that if v3 was tainted, then the first argument variable in the context of the called methods should be tainted as well. Since the called method is not static, the first argument variable p0 is always the reference of the current object. Then the register p1 in the context of the called method should also be tainted. On the other hand, whether the result of the function call v2 should be tainted also is undetermined at this juncture. Tainting in this case will depend on the content of the method, a potential problem. However, based on the "Reference Taint Propagation Rule" described in the next section, we will be able to determine unambiguously if v2 is tainted or not.

The rules that can be applied to cover the Dalvik instruction set are similar to those just described in the examples above, hence, we will not list the complete semantic table for all instructions here.

4.2.2 Reference Taint Propagation Rule

Designing the taint propagation rule for object references is somewhat different. A typical scenario in this case will look like the following:

aget vAA, vBB, vCC

The semantics of the instruction above are as follows: get value in array referenced by vBB at index vCC, and store in the value register at vAA. If vCC is tainted, we decide to taint vAA as well, although the data moved into vAA was not tainted. For our system to work, it is necessary to taint a value if the value is retrieved making use of a tainted index from a table that is. The rationale for this rule is as follows. Consider an instance of Malware who surreptitiously. may get a name from the "Android Contacts". The field name is of course tainted. Then, the same Malware can use that name as an index to the SMS database, retrieve a confidential SMS text message, and send it out. Clearly, the system must prevent such data ex-filtration. Therefore the *TrustDroid*^{TM} system must also taint the SMS text retrieved by the tainted name; otherwise there will be no warning when the SMS text is send out disclosing confidential information.

4.2.3 Cross Process taint propagation rule

In the Android environment. Android processes communicates with each via an Interprocess

Communication Mechanism (IPC), therefore one Android process could theoretically pass tainted tag resources to other processes via this IPC. In the Android environment, the IPC system is called Binder, and has been designed as a transparent system to the Dalvik Virtual Machine layer. Therefore, a process running upon on the Dalvik VM can call a remote method of another process just as it was calling a local one. In order to handle Android's IPC in Dalvik, we can extend the taint tracking mechanisms described above to handle the situation. Theoretically our basic approaches described above can be used for IPC taint propagation. A detail description of such mechanisms will be presented in a future manuscript together with our earlier research on this topic.

4.2.4 Native Library taint propagation rule

In the Android environment there are two kinds of libraries that *TrustDroid*^{TM} is required to handle properly. These are: (1) the system native library which is an element of the Dalvik virtual machine framework, and (2) user-imported libraries. These libraries are handled differently by *TrustDroid*^{TM}.

We will consider first the system native library that is part of Dalvik virtual machine framework. This native library is open-sourced, and it is distributed with the OS. Further, it does not change unless the Android OS is externally patched. In this case, all methods of the native library can be pre-processed using the *taint tracking system* previously described, and store the results in what we called the "known semantics library". Hence, when the *TrustDroid*^{TM} executes a call the resulting execution of the system native library could be semantically processed without delay while guaranteeing *taint tracking*. In this case, the accuracy of *taint tracking* is always guaranteed but more importantly, performance of the system does not suffer. Optimizing the performance of *TrustDroid*^{TM} is one of our major goals for the future. At this point, the proof of the *TrustDroid*^{TM} methodology took precedence.

The second case to be considered is that of user-imported libraries. Although our general design also works for C/C++ native code, the implementation would be totally different resulting from the fundamental difference between the languages and execution environments. According to statistics from [1], less than 4% of Android applications use JNI interfaces, and many of them are known as suspicious programs that deserve warning. Therefore for the time being, and as a way to deal with this issue, *TrustDroid*^{TM} will raise a warning flag when the system invokes a non-system JNI.

4.2.5 Secondary Storage taint propagation rule

At this point in time, the mechanism that *TrustDroid*^{TM} utilizes to handle the case when tainted data is stored in secondary storage, is simply to taint the entire unit. For example we taint an entire file if tainted data is written to that file. The tainted storage unit should be treated as one of the sensitive data sources in the future. We lose some granularity in this process, however the fact is that operations on files are often out of the monitoring capability of *TrustDroid*^{TM}, particularly when such files are accessible to other processes. Therefore by lowering our granularity we not only gain performance improvement but also reduce the risk of miss tracking.

This solution is rather coarse, and we are currently investigating other finer granularity approaches that do not affect the performance of *TrustDroid*^{TM}.

4.3 *TrustDroid*^{TM} Taint Tracking Engine

4.3.1 Overview

The taint tracking engine of *TrustDroid*^{TM} is composed of following four elements: (1) a source/sink definition set, (2) a file scanner, (3) a tag management system, and (4) the interface between this components. The file scanner scans the output file of the semantic analyzer. When it finds out data from a sensitive source, it taints the data and then tracks down through the call graph till the end or upon reaching a taint sink. During the tracking process, tainting/clear operations are sent to the tag management element through the interface. The file scanner could also scan from the sink side and track back, until it reaches a sensitive source.

The key to our design is the tag management system. The system must be able to emulate the existence of all data during execution time (recall *TrustDroid*™ performs static not run time analysis), and maintain tag information.

4.3.2 *TrustDroid*™ Tag Management System

As discussed in section 4.2.1, there are four types of data that need to be monitored: local variable, argument variable, class field and static field. Local and argument variables are temporary data that exist in registers while the other two type reside in memory. The monitor of Local and argument variables are independent of each thread while class fields and static field share the same process memory.

In order to analyze how tainted tag flows, *TrustDroid*™ must simulate an execution context for each thread of every process. As shown in section 2, Dalvik is a register based virtual machine, whereas each execution of a method declares a certain set of registers that it exclusively owns. These registers are used to store local variables and argument variables. This can be shown in our previous example by inspecting the jasmine text file below:

```
.method public onCreate(Landroid/os/Bundl
    .registers 6
    .parameter "savedInstanceState"
```

In this case, the method exclusively declares six-(6) registers, among which two are used to store parameters (one explicit parameter and another implicit). Hence, the context of a thread is essentially the set of registers it currently uses.

Our tag management uses a similar mechanism to that of a stack-based system, that is, it maintains a stack for each thread and creates a frame for each function call. What is put in the frame is the register set that is exclusively used by that particular function. Parameter passing is done inside the stack: caller leaves parameters in the stack so they contained the highest numbered registers of the called function. Actually, this is very similar to the way Dalvik maintains its internal stack. Note that *TrustDroid*™'s tag system does not need the value of the register but instead it requires the mapping of register and tag. Within this context, we will extend each register by one bit, containing a Boolean value, called it the bool bit.

As to track the tag of class field and static field, our tag management system keeps a data structure for each class instance. Within the each data structure, there is a mapping of index and tags, indicating tagging status of each field of the class. Again, the value of field is not needed, hence, the size of this data structure remains relatively small. The reference of a class instance point to somewhere in the stack while static class contains no reference.

5. TrustDroid™ Implementations & Deployment

In order to make TrustDroid™ work well, it should be properly protected, deployed and configured in an Android device.

Firstly, and in our current implementation the problem of protecting the *TrustDroid*™ itself is not addressed. We assume that an approach can be found to handle this problem. Instead, our approach relies on the integrity of both user space files and system space libraries.

A key issue that needs to be addressed is where in the process of running and Android application should *TrustDroid*™ be deployed. In our implementation, we have selected to deploy *TrustDroid*™ at the point where an application is loaded but not yet has begun execution. Such an approach will imply patching the Android O/S. Instead, and in order to facilitate the wide spread deployment of *TrustDroid*™, we have selected to have *TrustDroid*™ work as a common application that has the ability to accesses the file system and scan Apk files as scheduled.

Finally, *TrustDroid*™ success depends on the correct definition and configuration of sensitive data source and taint sink. In *TrustDroid*™ both sets are configurable and serve as an input to the system. Clearly, sensitive data sources usually contains but are not limited to: (1) information

rich databases, such as contacts; (2) device identifiers; (3) sensors such as GPS and camera.

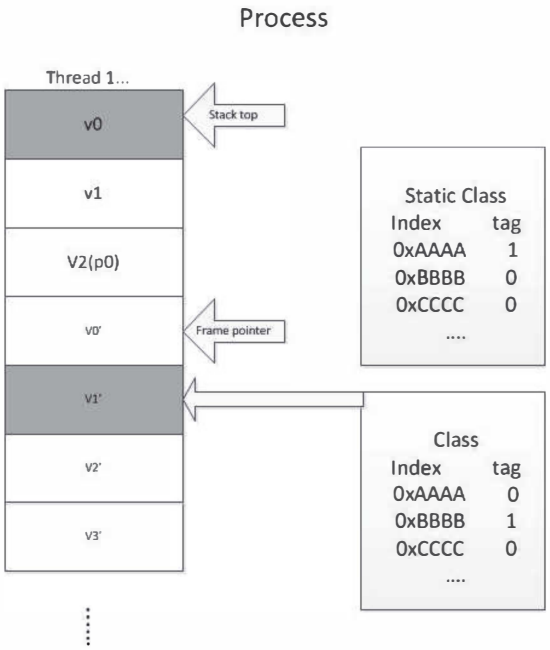

Figure 4 stack and class data structure. Dark grayed block in stack is tainted.

6. Related Work

Previously, Enck, et. al. describe a data flow tracking system called TaintDroid [1], which shares many similarities with *TrustDroid*™, However, there exists fundamental differences. At its core, TaintDroid uses dynamic taint checking by hooking the Interpreter in Dalvik and patching the memory system. This implies that in order for TaintDroid to operate correctly, a patched version of the Android O/S must be loaded, requiring a unique customized Android release. Such an approach creates a severe barrier for practical real-world deployment. *TrustDroid*™ works in a standalone in such a way that no patching of the Android O/S is required. Hence, *TrustDroid*™ could be deployed in real world situations transparently. Similar data flow tint tracking systems include ScanDroid, and Avik, see [9], [10]. However, in both cases, the approach requires the processing of source code which makes it highly unlikely that it could be deployed in a wide scale.

Researchers at several labs have taken a totally different approaches that involve the extension of Android's default permission system to prevent data ex-filtration. In this category, Nauman, et. al. [11] and Zhou et. al. [12], have extended the Android permission system to prevent abnormal access to sensitive data. However, due to the permission system's coarse granularity, their approach lacks the resolution needed to prevent sensitive data ex-filtration. Other researches such as Ongtang et. al., see[13], have developed an approach where as the security of an application is analyzed based on the access policies of the system. In this approach, just like in [11], the main limitation is the granularity/resolution of the protection provided. In addition researchers at the Institute for Applied Information Processing and Communications (IAIK), see Winter [8], have exploited recent additions to the ARM architecture implementing the Trusted Computing environment in a mobile platform. The approach is based on the idea of the simultaneous creation of a Trusted and Un-trusted zone utilizing the ARM architecture special memory management registers and environment. In this approach, and unlike previous attempts to build a secure kernel in the Linux environment, the Zones are managed by software rather than firmware leading to the usual concerns while implementing a secure O/S. Finally, Nauman et.al., see [7] presents an architecture to potentially create a remote attestation service , called TC, that allows a service provider or a device owner to determine whether the device is in a trusted state before releasing protected data to or storing private information on the phone. The basic limitation of this approach for real case applications is the fact that it requires a trusted chain for the trusted computing environment and remote attestation. Given the current state of the Android platform such an approach is not realistic. However, we believe that a combination of the approaches described in [7] and [8] can served as the basis for the creation of a Trusted Mobile device.

7. TrustDroid™ limitations and Future Work

As a static analyzer, a significant defect of the approach presented here is the ability to deal with the dynamic execution loaded code. As we know, Java provides an interface to dynamically load stream into memory and execute that stream. In

particular, the stream could be encrypted before being loaded. The ability to perform semantic analysis on an encrypted stream is hard, however, one can consider the action of loading encrypted stream itself as suspicious. Such an approach is limiting. Hence, in future implementations of *TrustDroid*^{TM}, this issue will be addressed.

Finally, a key limitation of *TrustDroid*^{TM} is inability to support the JNI interface, as discussed in 4.2.4. This is currently being addressed and together with a detailed evaluation of our *TrustDroid*^{TM} implementation in a real environment will appear in a future manuscript.

8. References

[1] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, Anmol N. Sheth. *"TaintDroid: An Information-Flow Tracking System for Real-time Privacy Monitoring on Smartphones"* To appear at the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)

[2] DEX File Format http://source.android.com/tech/Dalvik/dex-format.html

[3] ANTLR http://www.antlr.org/

[4] Dalvik opcode http://www.netmite.com/android/mydroid/Dalvik/docs/Dalvik-bytecode.html

[5] Jasmin, http://jasmin.sourceforge.net/guide.html

[6] Smali http://code.google.com/p/smali/

[7] Mohammad Nauman, Sohail Khan, Xinwen Zhang and Jean-Pierre Seifert, *"Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform"*,TRUST'10 Process Proceedings of the 3rd international conference on Trust and trustworthy computing Pages 1-15

[8] Johannes Winter, *"Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms"*, STC '08: Proceedings of the 3rd ACM workshop on Scalable

[9] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster, *"SCanDroid: Automated Security Certification of Android Applications"*, In: 5th ACM Symposium on Information, Computer and Communications Security. (2010)

[10] Avik Chaudhuri, *"Language-Based Security on Android PLAS"*, '09 Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security

[11] Mohammad Nauman, Sohail Khan, Xinwen Zhang, *"Apex: extending Android permission model and enforcement with user-defined runtime constraints"*, ASIACCS '10 Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security

[12] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, Vincent W. Freeh, *"Taming Information-Stealing Smartphone Applications (on Android)"*, **TRUST'11:** Proceedings of the 4th international conference on Trust and trustworthy computing

[13] Machigar Ongtang, Stephen McLaughlin, William Enck, Patrick McDaniel, *"Semantically Rich Application-Centric Security"*, in Android Security and Communication Networks , Volume 5 Issue , John Wiley & Sons, Inc.

[14] H Yin, D Song, M Egele, C Kruegel, E Kirda, *"Panorama: Capturing System-Wide InformationFlow for Malware Detection and Analysis"*. CCS '07 Proceedings of the 14th ACM conference on Computer and communications security

[15] Trusted Computing Group, http://www.trustedcomputinggroup.org/