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

Golam Sarwar (Babil),[1,2] Olivier Mehani,[1] Roksana Boreli,[1,2] and Mohamed-Ali Kaafar[1,3]
[1]*NICTA, Eveleigh, Sydney, NSW, Australia*
[2]*UNSW, Kensington, Sydney, NSW, Australia*
[3]*Inria, Grenoble, Rhône-Alpes, France*

Keywords: Dynamic Taint Analysis, Privacy, Malware, Anti-Taint-Analysis, Anti-TaintDroid, Android.

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 methods in a mobile device environment to circumvent this security technique. 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 evaluate 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.

1 INTRODUCTION

Mobile devices have become an integral part of our daily lives, with hugely increased usage of various applications and services 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 (Egele et al., 2011; Hornyack et al., 2011). Collecting information from user's mobile devices has actually become a line of business (*e.g.*, 201, 2011). Such data may be used for a number of purposes, ranging from identity theft to profiling and tracking for purposes of targeted advertising (Grace et al., 2012).

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 (Felt et al., 2011), and users are usually unable to properly evaluate these requests (Felt et al., 2012). 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 users' data are required. A number of tools to achieve this have been developed in recent years.[2] Within the research community, the TaintDroid (Enck et al., 2012) tool has received a lot of attention and a num-

*This paper is a shortened version of the technical report available at http://www.nicta.com.au/pub?id=7091

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

Sarwar G., Mehani O., Boreli R. and Kaafar M..
On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices.
DOI: 10.5220/0004535104610468
In *Proceedings of the 10th International Conference on Security and Cryptography* (SECRYPT-2013), pages 461-468
ISBN: 978-989-8565-73-0

ber of extensions have also been proposed and implemented (Hornyack et al., 2011; Russello et al., 2012). This patch for the Android system uses dynamic taint analysis (Newsome and Song, 2005; Schwartz et al., 2010) to track sensitive data as it is used by (untrusted) apps. It "taints" sensitive data, and warns the user when these variables are leaked.

Prior work on taint analysis has already identified both conceptual and technical limitations (Cavallaro et al., 2007, 2008; Schwartz et al., 2010), that can be exploited to avoid detection. Dynamic anti-taint techniques have been classified by Cavallaro et al. (2008).

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. Our focus here is on dynamic taint analysis and that the use of static analysis, which is sometimes suggested as a complementary technique in these contexts (*e.g.*, Graa et al., 2012), 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 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. Finally, we characterise 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

2.1 Taint Tracking

Taint analysis was originally proposed as a method to track the lifetime of data in a program (Chow et al., 2004). 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 (Newsome and Song, 2005) is an extension of the technique to perform this data tracking in real-time, as the program is executed. Taint tracking mechanisms have been implemented in a number of programming languages (*e.g.* Thomas and Hunt, 2001; 201, 2012), as a way to support the developer's task of writing valid code.

More recently, the use of the technique has seen a renewed interest for malware analysis and detection. Ho et al. (2006) 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 (Yin et al., 2007), flags potentially malicious code by identifying how it uses sensitive data it captures. 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 (Enck et al., 2012), or even earlier on in the App markets, with AppInspector (Gilbert et al., 2011).

A noteworthy property of this second class of 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, previously untrusted, is now the item to protect.

2.2 TaintDroid

TaintDroid (Enck et al., 2012) is an implementation of dynamic taint analysis for the Android platform.

It is implemented as an extension to the Dalvik virtual machine, and can oversee all activity which runs above it.

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 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 code, it shares the limitations of dynamic taint analysis (Cavallaro et al., 2007, 2008; Schwartz et al., 2010). Enck et al. (2012) acknowledge that TaintDroid is vulnerable to control dependence attacks as well as some side-channel attacks. Nonetheless, user data-protection solutions like AppFence (Hornyack et al., 2011) and MOSES (Russello et al., 2012) 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 1. 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 user (step 1), and do not consider potential infection vectors. However, we also assume this user 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.

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.

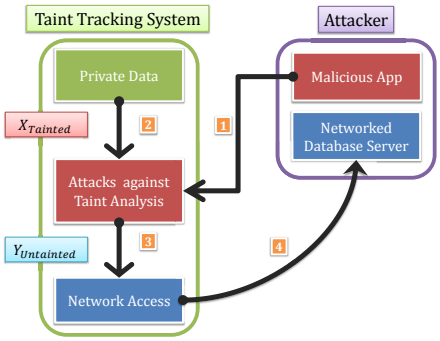

Figure 1: Our attack model against dynamic taint analysis used for detection of malware leaking sensitive information.

4 ANTI-TAINT-ANALYSIS TECHNIQUES

In this section we introduce the generic classes of attacks against taint-based data leak protection. In the following, we assume that X_{Tainted} is a single byte, however, the attacks presented are applicable to any type of data.

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 (Newsome and Song, 2005; Enck et al., 2012), 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 $Y_{\text{Untainted}}$.

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 where a value matching X_{Tainted} 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.

Algorithm 1: Simple Encoding Attack.

```
for each symbol ∈ AsciiTable do
    if symbol = X_Tainted then
        Y_Untainted ← symbol
    end if
end for
```

4.1.2 Count-to-X Attack

Instead of traversing an array in search for the value related to X_{Tainted}, the count-to-X attack recreates the value one incrementation at a time, until $Y_{\text{Untainted}}$ matches X_{Tainted}.

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. It can, for example, keep count of how many times it has been called as the representation of X_{Tainted}.

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 $Y_{\text{Untainted}}$.

4.2.1 System Command Attack

It is possible to leverage system 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 $Y_{\text{Untainted}}$.

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 more than 40 executables to be vulnerable for this kind of attack. 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.

As described by Enck et al. (2012), 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 $Y_{\text{Untainted}}$.

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. Such medium is often overlooked by taint-checking mechanisms, and not effectively protected. These attacks might be the hardest to protect against as they cover the entire system.

4.3.1 Timing Attack

Timing attacks rely on the specific side channel created by the time it takes to perform some task. They can be performed from within a program actively trying to leak tainted data by using delay loops with a variable duration depending on the value of a tainted variable. They are 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.

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 $Y_{\text{Untainted}} = X_{\text{Tainted}}$ 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 2, 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 2: File Length Attack.

$F \leftarrow CreateNewFileHandle()$
$z \leftarrow 0$
while $z < X_{\text{Tainted}}$ **do**
  $WriteOneByte(F)$
  $z \leftarrow z+1$
end while
$Y_{\text{Untainted}} \leftarrow ReadFileLength(F)$

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.

If the system provides a clipboard for applications to store and exchange temporary data, a very similar technique can be used: the **Clipboard Length Attack**.

4.3.3 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 OCR techniques.

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. 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 3, which modifies the arbitrarily chosen pixel at coordinates 10×10.

Algorithm 3: Bitmap Pixel Attack.

$B \leftarrow CreateNewBitmap()$
// set the pixel at coordinate (10, 10) with X_{Tainted}
$SetPixel([10,10], X_{\text{Tainted}} \rightarrow B)$
$Y_{\text{Untainted}} \leftarrow GetPixel(B, [10,10])$

4.3.4 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 4 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 4: Text Scaling Attack.

$T \leftarrow TextViewWidget()$
$T \leftarrow SetTextScalingValue(X_{\text{Tainted}})$
$Y_{\text{Untainted}} \leftarrow GetTextScalingValue(T)$

4.3.5 Direct Buffer Attack

Pointer indirection attacks target the low level memory access features of the system. In this particular attack, shown in Algorithm 5, 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 5: Direct Buffer Attack.

$D \leftarrow NewDirectAccessBuffer()$
// write X_{Tainted} *at location* $0\times XX$ *of buffer D*
$DirectMemoryWrite(X_{\text{Tainted}}, 0\times XX \rightarrow D)$
// read from memory location $0\times XX$ *of buffer D*
$Y_{\text{Untainted}} \leftarrow DirectMemoryRead(D, 0\times 00)$

In ScrubDroid, this attack works due to an implementation limitation of TaintDroid that has been mentioned by Enck et al. (2012). We include this attack in-line with the classification of Cavallaro et al. (2008) 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 (Google Inc., 2012) 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,[3] 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.

Our experimental framework is as follows. For each attack, we first query non-sensitive (untainted) information. We then query for 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, 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, we evaluate how practical it is for the attacker to conduct the various proposed attacks by measuring 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 1(a), 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 1(b), 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 POTENTIAL COUNTER MEASURES AND DISCUSSION

Clause et al. (2007); Kang et al. (2011) have proposed techniques to fight **control dependence attacks** by over-marking all the variables involved in conditional statements. 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 Clause et al., 2007; Kang et al., 2011) are

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

Table 1: 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

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 (201, 2012). 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 Hornyack et al., 2011 or MOSES Russello et al., 2012), 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 4.3.5, 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 TaintDroid 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 alluded to by Kang et al. (2011), 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.

7 CONCLUSIONS

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.

REFERENCES

(2011). Understanding Carrier IQ technology. White paper, Carrier IQ.

(2012). *perlsec - Perl security*.

Cavallaro, L., Saxena, P., and Sekar, R. (2007). Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. Technical report, Stony Brook University.

Cavallaro, L., Saxena, P., and Sekar, R. (2008). On the limits of information flow techniques for malware analysis and containment detection of intrusions and malware, and vulnerability assessment. In *DIMVA 2008*, chapter 8.

Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. (2004). Understanding data lifetime via whole system simulation. In *Security 2004*.

Clause, J., Li, W., and Orso, A. (2007). Dytan: a generic dynamic taint analysis framework. In *ISTA 2007*.

Egele, M., Kruegel, C., Kirda, E., and Vigna, G. (2011). PiOS: Detecting privacy leaks in iOS applications. In *NDSS 2011*.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2012). TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI 2010*.

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011). Android permissions demystified. In *CCS 2011*.

Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. (2012). Android permissions: User attention, comprehension, and behavior. In *SOUPS 2012*.

Gilbert, P., Chun, B. G., Cox, L. P., and Jung, J. (2011). Vision: Automated security validation of mobile apps at app markets. In *MCS 2011*.

Google Inc. (2012). Android Java New I/O interface. Android 4.2 r1.

Graa, M., Cuppens-Boulahia, N., Cuppens, F., and Cavalli, A. (2012). Detecting control flow in smarphones: Combining static and dynamic analyses. In *CCS 2012*.

Grace, M. C., Zhou, W., Jiang, X., and Sadeghi, A.-R. (2012). Unsafe exposure analysis of mobile in-app advertisements. In *WiSec 2012*.

Ho, A., Fetterman, M., Clark, C., Warfield, A., and Hand, S. (2006). Practical taint-based protection using demand emulation. In *EuroSys 2006*.

Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. (2011). "These aren't the droids you're looking for:" retrofitting Android to protect data from imperious applications. In *CCS 2011*.

Kang, M. G., McCamant, S., Poosankam, P., and Ong, D. (2011). DTA++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS 2011*.

Newsome, J. and Song, D. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS 2005*.

Russello, G., Conti, M., Crispo, B., and Fernandes, E. (2012). MOSES: Supporting operation modes on smartphones. In *SACMAT 2012*.

Schwartz, E. J., Avgerinos, T., and Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *SP 2010*.

Thomas, D. and Hunt, A. (2001). *Locking Ruby in the Safe*, chapter 20.

Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. (2007). Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS 2007*.